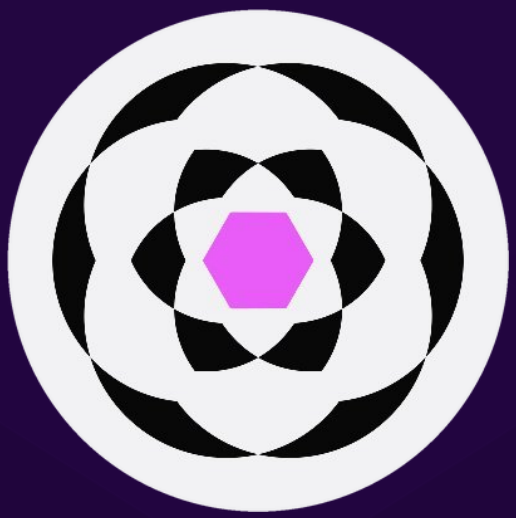
SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for: Perennial
Prepared by: Sherlock
Lead Security Expert: panprog
Dates Audited: March 11 - April 6, 2024
Prepared on: April 23, 2024

Introduction

Perennial is a DeFi-native derivatives primitive that allows for the creation of two-sided markets that trade exposure to an underlying price feed in a capital efficient manner.

Scope

Repository: equilibria-xyz/perennial-v2

Branch: v2.2

Commit: 22ba19c323a13c9f02f95db6747d137a3bf1277a

Repository: equilibria-xyz/root

Branch: v2.2

Commit: 8faa77e920c23b67f12942ec61b2580f0533d161

Repository: equilibria-xyz/emptyset-mono

Branch: britz-migration-reserve

Commit: ab24b00bff3d9a3416e036ccb1cad09dfe3f005a

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
10	3

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: Empty orders do not request from oracle and during settlement they use an invalid oracle version with `price=0` which messes up a lot of fees and funding accounting leading to loss of funds for the makers

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/5

Found by

panprog

Summary

When `market.update` which doesn't change user's position is called, a new (current) global order is created, but the oracle version is not requested due to empty order. This means that during the order settlement, it will use non-existant invalid oracle version with `price = 0`. This price is then used to accumulate all the data in this invalid `Version`, meaning accounting is done using `price = 0`, which is totally incorrect. For instance, all funding and fees calculations multiply by oracle version's price, thus all time periods between empty order and the next valid oracle version will not accumulate any fees, which is funds usually lost by makers (as makers won't receive fees/funding for the risk they take).

Vulnerability Detail

When `market.update` is called, it requests a new oracle version at the current order's timestamp unless the order is empty:

```
// request version
if (!newOrder.isEmpty()) oracle.request(IMarket(this), account);
```

The order is empty when it doesn't modify user position:

```
function isEmpty(Order memory self) internal pure returns (bool) {
    return pos(self).isZero() && neg(self).isZero();
}

function pos(Order memory self) internal pure returns (UFixed6) {
    return self.makerPos.add(self.longPos).add(self.shortPos);
}

function neg(Order memory self) internal pure returns (UFixed6) {
```

```
    return self.makerNeg.add(self.longNeg).add(self.shortNeg);
}
```

Later, when a valid oracle version is commited, during the settlement process, oracle version at the position is used:

```
function _processOrderGlobal(
    Context memory context,
    SettlementContext memory settlementContext,
    uint256 newOrderId,
    Order memory newOrder
) private {
    // @audit no oracle version at this timestamp, thus it's invalid with
↪   `price=0`
    OracleVersion memory oracleVersion = oracle.at(newOrder.timestamp);

    context.pending.global.sub(newOrder);
    // @audit order is invalidated (it's already empty anyway), but the
↪   `price=0` is still used everywhere
    if (!oracleVersion.valid) newOrder.invalidate();

    VersionAccumulationResult memory accumulationResult;
    (settlementContext.latestVersion, context.global, accumulationResult) =
↪   VersionLib.accumulate(
        settlementContext.latestVersion,
        context.global,
        context.latestPosition.global,
        newOrder,
        settlementContext.orderOracleVersion,
        oracleVersion, // @audit <<< when oracleVersion is invalid, the
↪   `price=0` will still be used here
        context.marketParameter,
        context.riskParameter
    );
...
```

If the oracle version is invalid, the order is invalidated, but the `price=0` is still used to accumulate. It doesn't affect pnl from price move, because the final oracle version is always valid, thus the correct price is used to evaluate all possible account actions, however it does affect accumulated fees and funding:

```
function _accumulateLinearFee(
    Version memory next,
    AccumulationContext memory context,
    VersionAccumulationResult memory result
) private pure {
```


```
    (UFixed6 makerLinearFee, UFixed6 makerSubtractiveFee) =
↪   _accumulateSubtractiveFee(
        context.riskParameter.makerFee.linear(
            Fixed6Lib.from(context.order.makerTotal()),
            context.toOracleVersion.price.abs() // @audit <<< price == 0 for
↪   invalid oracle version
        ),
        context.order.makerTotal(),
        context.order.makerReferral,
        next.makerLinearFee
    );
...
    // Compute long-short funding rate
    Fixed6 funding = context.global.pAccumulator.accumulate(
        context.riskParameter.pController,
        toSkew.unsafeDiv(Fixed6Lib.from(context.riskParameter.takerFee.scale())).m
↪   in(Fixed6Lib.ONE).max(Fixed6Lib.NEG_ONE),
        context.fromOracleVersion.timestamp,
        context.toOracleVersion.timestamp,
        context.fromPosition.takerSocialized().mul(context.fromOracleVersion.pri
↪   ce.abs()) // @audit <<< price == 0 for invalid oracle version
    );
...
function _accumulateInterest(
    Version memory next,
    AccumulationContext memory context
) private pure returns (Fixed6 interestMaker, Fixed6 interestLong, Fixed6
↪   interestShort, UFixed6 interestFee) {
    // @audit price = 0 and notional = 0 for invalid oracle version
    UFixed6 notional = context.fromPosition.long.add(context.fromPosition.short)
↪   .min(context.fromPosition.maker).mul(context.fromOracleVersion.price.abs());
...
```

As can be seen, all funding and fees accumulations multiply by oracle version's price (which is 0), thus during these time intervals fees and funding are 0.

This will happen by itself during **any** period when there are no orders, because oracle provider's settlement callback uses `market.update` with empty order to settle user account, thus any non-empty order is always followed by an empty order for the next version and `price = 0` will be used to settle it until the next non-empty order:

```
function _settle(IMarket market, address account) private {
    market.update(account, UFixed6Lib.MAX, UFixed6Lib.MAX, UFixed6Lib.MAX,
↪   Fixed6Lib.ZERO, false);
}
```

Impact

All fees and funding are incorrectly calculated as 0 during any period when there are no non-empty orders (which will be substantially more than 50% of the time, more like 90% of the time). Since most fees and funding are received by makers as a compensation for their price risk, this means makers will lose all these under-calculated fees and will receive a lot less fees and funding than expected.

Proof of concept

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

```
it('no fees accumulation due to invalid version with price = 0', async () => {

function setupOracle(price: string, timestamp : number, nextTimestamp : number) {
    const oracleVersion = {
    price: parse6decimal(price),
    timestamp: timestamp,
    valid: true,
    }
    oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
    oracle.status.returns([oracleVersion, nextTimestamp])
    oracle.request.returns()
}

function setupOracleAt(price: string, valid : boolean, timestamp : number) {
    const oracleVersion = {
    price: parse6decimal(price),
    timestamp: timestamp,
    valid: valid,
    }
    oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
}

const riskParameter = { ...(await market.riskParameter()) }
const riskParameterMakerFee = { ...riskParameter.makerFee }
riskParameterMakerFee.linearFee = parse6decimal('0.005')
riskParameterMakerFee.proportionalFee = parse6decimal('0.0025')
riskParameterMakerFee.adiabaticFee = parse6decimal('0.01')
riskParameter.makerFee = riskParameterMakerFee
const riskParameterTakerFee = { ...riskParameter.takerFee }
riskParameterTakerFee.linearFee = parse6decimal('0.005')
riskParameterTakerFee.proportionalFee = parse6decimal('0.0025')
riskParameterTakerFee.adiabaticFee = parse6decimal('0.01')
riskParameter.takerFee = riskParameterTakerFee
await market.connect(owner).updateRiskParameter(riskParameter)
```

```
dsu.transferFrom.whenCalledWith(user.address, market.address,
↪  COLLATERAL.mul(1e12)).returns(true)
dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪  COLLATERAL.mul(1e12)).returns(true)

setupOracle('100', TIMESTAMP, TIMESTAMP + 100);

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

setupOracle('100', TIMESTAMP + 100, TIMESTAMP + 200);
await market
    .connect(user)
    ['update(address,uint256,uint256,uint256,int256,bool)'](user.address,
↪  POSITION, 0, 0, 0, false);

// oracle is commited at timestamp+200
setupOracle('100', TIMESTAMP + 200, TIMESTAMP + 300);
await market
    .connect(user)
    ['update(address,uint256,uint256,uint256,int256,bool)'](user.address,
↪  POSITION, 0, 0, 0, false);

// oracle is not commited at timestamp+300
setupOracle('100', TIMESTAMP + 200, TIMESTAMP + 400);
setupOracleAt('0', false, TIMESTAMP + 300);
await market
    .connect(user)
    ['update(address,uint256,uint256,uint256,int256,bool)'](user.address,
↪  POSITION, 0, 0, 0, false);

// settle to see makerValue at all versions
setupOracle('100', TIMESTAMP + 400, TIMESTAMP + 500);

await market.settle(user.address);
await market.settle(userB.address);

var ver = await market.versions(TIMESTAMP + 200);
console.log("version 200: longValue: " + ver.longValue + " makerValue: " +
↪  ver.makerValue);
```

```
var ver = await market.versions(TIMESTAMP + 300);
console.log("version 300: longValue: " + ver.longValue + " makerValue: " +
↪  ver.makerValue);
var ver = await market.versions(TIMESTAMP + 400);
console.log("version 400: longValue: " + ver.longValue + " makerValue: " +
↪  ver.makerValue);
})
```

Console log:

```
version 200: longValue: -318 makerValue: 285
version 300: longValue: -100000637 makerValue: 100500571
version 400: longValue: -637 makerValue: 571
```

Notice, that fees are accumulated between versions 200 and 300, version 300 has huge pnl (because it's evaluated at price = 0), which then returns to normal at version 400, but no fees are accumulated between version 300 and 400 due to version 300 having `price = 0`.

Code Snippet

`Market._update` requests a new oracle version only when the order is not empty: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L466-L467

`Market._processOrderGlobal` invalidates the order for invalid oracle version, but still uses invalid oracle's price (which is 0) to accumulate: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L610-L625

Tool used

Manual Review

Recommendation

Keep the price from the previous valid oracle version and use it instead of oracle version's one if oracle version's price == 0.

Discussion

sherlock-admin2

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

panprog commented:

invalid by sherlock rules

nevillehuang

@panprog What is your comment referring to here?

panprog

@nevillehuang The comment was meant for issue #7, somehow got mixed up with this one. This issue is valid.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/301

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-2: Requested oracle versions, which have expired, must return this oracle version as invalid, but they return it as a normal version with previous version's price instead

Source:
https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/6

Found by

bin2chen, panprog

Summary

Each market action requests a new oracle version which must be commited by the keepers. However, if keepers are unable to commit requested version's price (for example, no price is available at the time interval, network or keepers are down), then after a certain timeout this oracle version will be commited as invalid, using the previous valid version's price.

The issue is that when this expired oracle version is used by the market (using `oracle.at`), the version returned will be valid (`valid = true`), because oracle returns version as invalid only if `price = 0`, but the `commit` function sets the previous version's price for these, thus it's not 0.

This leads to market using invalid versions as if they're valid, keeping the orders (instead of invalidating them), which is a broken core functionality and a security risk for the protocol.

Vulnerability Detail

When requested oracle version is commited, but is expired (commited after a certain timeout), the price of the previous valid version is set to this expired oracle version:

```
function _commitRequested(OracleVersion memory version) private returns (bool) {
    if (block.timestamp <= (next() + timeout)) {
        if (!version.valid) revert KeeperOracleInvalidPriceError();
        _prices[version.timestamp] = version.price;
    } else {
        // @audit previous valid version's price is set for expired version
        _prices[version.timestamp] = _prices[_global.latestVersion];
    }
    _global.latestIndex++;
```

```
    return true;
}
```

Later, `Market._processOrderGlobal` reads the oracle version using the `oracle.at`, invalidating the order if the version is invalid:

```
function _processOrderGlobal(
    Context memory context,
    SettlementContext memory settlementContext,
    uint256 newOrderId,
    Order memory newOrder
) private {
    OracleVersion memory oracleVersion = oracle.at(newOrder.timestamp);

    context.pending.global.sub(newOrder);
    if (!oracleVersion.valid) newOrder.invalidate();
```

However, expired oracle version will return `valid = true`, because this flag is only set to `false` if `price = 0`:

```
function at(uint256 timestamp) public view returns (OracleVersion memory
↪   oracleVersion) {
    (oracleVersion.timestamp, oracleVersion.price) = (timestamp,
↪   _prices[timestamp]);
    oracleVersion.valid = !oracleVersion.price.isZero(); // @audit <<< valid =
↪   false only if price = 0
}
```

This means that `_processOrderGlobal` will treat this expired oracle version as valid and won't invalidate the order.

Impact

Market uses invalid (expired) oracle versions as if they're valid, keeping the orders (instead of invalidating them), which is a broken core functionality and a security risk for the protocol.

Code Snippet

`KeeperOracle._commitRequested` sets `_prices` to the last valid version's price for expired versions: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-oracle/contracts/keeper/KeeperOracle.sol#L153-L162

`Market._processOrderGlobal` reads the oracle version using the `oracle.at`,

invalidating the order if the version is invalid: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L604-L613

`KeeperOracle.at` returns `valid = false` only if `price = 0`, but since expired version has valid price, it will be returned as a valid version: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-oracle/contracts/keeper/KeeperOracle.sol#L109-L112

Tool used

Manual Review

Recommendation

Add validity map along with the price map to `KeeperOracle` when recording commited price.

Discussion

nevillehuang

@arjun-io @panprog @bin2chen66 For the current supported tokens in READ.ME, I think medium severity remains appropriate given they are both stablecoins. Do you agree?

arjun-io

> @arjun-io @panprog @bin2chen66 For the current supported tokens in READ.ME, I think medium severity remains appropriate given they are both stablecoins. Do you agree?

I'm not entirely sure how the stablecoin in use matters here? Returning an invalid versions as valid can be very detrimental in markets where invalid versions can be triggered at will (such as in markets that close) which can result in users being able to open or close positions when they shouldn't be able to

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/equilibria-xyz/perennial-v2/pull/308

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-3: Vault global shares and assets change will mismatch local shares and assets change during settlement due to incorrect `_withoutSettlementFeeGlobal` formula

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/26

Found by

panprog

Summary

Every vault update, which involves change of position in the underlying markets, `settlementFee` is charged by the Market. Since many users can deposit and redeem during the same oracle version, this `settlementFee` is shared equally between all users of the same oracle version. However, there is an issue in that `settlementFee` is charged once both for deposits and redeems, however `_withoutSettlementFeeGlobal` subtracts `settlementFee` in full both for deposits and redeems, meaning that for global fee, it's basically subtracted twice (once for deposits, and another time for redeems). But for local fee, it's subtracted proportional to `checkpoint.orders`, with sum of fee subtracted equal to exactly `settlementFee` (once). This difference in global and local `settlementFee` calculations leads to inflated `shares` and `assets` added for user deposits (local state) compared to vault overall (global state).

Vulnerability Detail

Here is an easy scenario to demonstrate the issue:

1. `SettlementFee = $10`
2. User1 deposits `$10` for oracle version `t = 100`
3. User2 redeems `10 shares` (worth `$10`) for the same oracle version `t = 100` (`checkpoint.orders = 2`)
4. Once the oracle version `t = 100` settles, we have the following: 4.1. Global deposits = $10, redeems = $10 4.2. Global deposits convert to `0 shares` (because `_withoutSettlementFeeGlobal(10)` applies `settlementFee` of $10 in full, returning `10-10=0`) 4.3. Global redeems convert to `0 assets` (because `_withoutSettlementFeeGlobal(10)` applies `settlementFee` of $10 in full, returning `10-10=0`) 4.4. User1 deposit of $10 converts to `5 shares` (because `_withoutSettlementFeeLocal(10)` applies `settlementFee` of

$5 (because there are 2 orders), returning `10 - 5 = 5`) 4.5. User 2 redeem of 10 shares converts to `5` (for the same reason)

From the example above it can be seen that:

1. User1 receives 5 shares, but global vault shares didn't increase. Over time this difference will keep growing potentially leading to a situation where many user redeems will lead to 0 global shares, but many users will still have local shares which they will be unable to redeem due to underflow, thus losing funds.
2. User2's assets which he can claim increase by $5, but global claimable assets didn't change, meaning User2 will be unable to claim assets due to underflow when trying to decrease global assets, leading to loss of funds for User2.

The underflow in both cases will happen in `Vault._update` when trying to update global account:

```
function update(
    Account memory self,
    uint256 currentId,
    UFixed6 assets,
    UFixed6 shares,
    UFixed6 deposit,
    UFixed6 redemption
) internal pure {
    self.current = currentId;
    // @audit global account will have less assets and shares than sum of local
    ↪  accounts
    (self.assets, self.shares) = (self.assets.sub(assets),
    ↪  self.shares.sub(shares));
    (self.deposit, self.redemption) = (self.deposit.add(deposit),
    ↪  self.redemption.add(redemption));
}
```

Impact

Any time there are both deposits and redeems in the same oracle version, the users receive more (local) shares and assets than overall vault shares and assets increase (global). This mismatch causes:

1. Systematic increase of (sum of user shares - global shares), which can lead to bank run since the last users who try to redeem will be unable to do so due to underflow.
2. Systematic increase of (sum of user assets - global assets), which will lead to users being unable to claim their redeemed assets due to underflow.

The total difference in local and global `shares+assets` equals to `settlementFee` per each oracle version with both deposits and redeems. This can add up to significant amounts (at `settlementFee = $1` this can be $100-1000$ per day), meaning it will quickly become visible especially for point 2., because typically global claimable assets are at or near 0 most of the time, since users usually redeem and then immediately claim, thus any difference of global and local assets will quickly lead to users being unable to claim.

Code Snippet

SettlementFee subtracted in `_withoutSettlementFeeGlobal`
https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/types/Checkpoint.sol#L183-L185

This is subtracted twice: for deposit and for redeem:
https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/types/Account.sol#L62-L63

Tool used

Manual Review

Recommendation

Calculate total orders to deposit and total orders to redeem (in addition to total orders overall). Then `settlementFee` should be multiplied by `deposit/orders` for `toGlobalShares` and by `redeems/orders` for `toGlobalAssets`. This weightening of `settlementFee` will make it in-line with local order weights.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/305

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: When vault's market weight is set to 0 to remove the market from the vault, vault's leverage in this market is immediately set to max leverage risking position liquidation

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/8

The protocol has acknowledged this issue.

Found by

panprog

Summary

If any market has to be removed from the vault, the only way to do this is via setting this market's weight to 0. The problem is that the first vault rebalance will immediately withdraw max possible collateral from this market, leaving vault's leverage at max possible leverage, risking the vault's position liquidation. This is especially dangerous if vault's position in this removed market can not be closed due to high skew, so min position is not 0, but the leverage will be at max possible value. As a result, vault depositors can lose funds due to liquidation of vault's position in this market.

Vulnerability Detail

When vault is rebalanced, each market's collateral is calculated as following:

```
marketCollateral = marketContext.margin
    .add(collateral.sub(totalMargin).mul(marketContext.registration.weight));

UFixed6 marketAssets = assets
    .mul(marketContext.registration.weight)
    .min(marketCollateral.mul(LEVERAGE_BUFFER));
```

For removed markets (`weight = 0`), `marketCollateral` will be set to `marketContext.margin` (i.e. minimum valid collateral to have position at max leverage), `marketAssets` will be set to 0. But later the position will be adjusted in case minPosition is not 0:

```
target.position = marketAssets
    .muldiv(marketContext.registration.leverage, marketContext.latestPrice.abs())
```

```
    .max(marketContext.minPosition)
    .min(marketContext.maxPosition);
```

This means that vault's position in the market with weight 0 will be at max leverage until liquidated or position can be closed.

Impact

Market removed from the vault (weight set to 0) is put at max leverage and has a high risk of being liquidated, thus losing vault depositors funds.

Proof of concept

The scenario above is demonstrated in the test, change the following test in test/integration/vault/Vault.test.ts:

```
    it('simple deposits and redemptions', async () => {
...
      // Now we should have opened positions.
      // The positions should be equal to (smallDeposit + largeDeposit) *
↪  leverage originalOraclePrice.
      expect(await position()).to.equal(
        smallDeposit.add(largeDeposit).mul(leverage).mul(4).div(5).div(originalO
↪  raclePrice),
      )
      expect(await btcPosition()).to.equal(
        smallDeposit.add(largeDeposit).mul(leverage).div(5).div(btcOriginalOracl
↪  ePrice),
      )

      /*** remove all lines after this and replace with the following code: ***/

      console.log("pos1 = " + (await position()) + " pos2 = " + (await
↪  btcPosition()) + " col1 = " + (await collateralInVault()) + " col2 = " +
↪  (await btcCollateralInVault()));

      // update weight
      await vault.connect(owner).updateWeights([parse6decimal('1.0'),
↪  parse6decimal('0')])

      // do small withdrawal to trigger rebalance
      await vault.connect(user).update(user.address, 0, smallDeposit, 0)
      await updateOracle()
```

```
      console.log("pos1 = " + (await position()) + " pos2 = " + (await
↪  btcPosition()) + " col1 = " + (await collateralInVault()) + " col2 = " +
↪  (await btcCollateralInVault()));
    })
```

Console log:

```
pos1 = 12224846 pos2 = 206187 col1 = 8008000000 col2 = 2002000000
pos1 = 12224846 pos2 = 206187 col1 = 9209203452 col2 = 800796548
```

Notice, that after rebalance, position in the removed market (pos2) is still the same, but the collateral (col2) reduced to minimum allowed.

Code Snippet

Vault market allocation sets collateral to only the margin if `weight = 0`: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L152-L153

Tool used

Manual Review

Recommendation

Ensure that the market's collateral is based on leverage even if `weight = 0`

Discussion

arjun-io

We would likely consider this a low for two reason:

1. The admin has control over when the vault is set to 0 weight, so as long as the limits don't prevent the vault from fully closing the position this is a safe operation to perform
2. The liquidation fee would be an acceptable cost in the cases when the above doesn't apply (i.e. in an emergency)

nevillehuang

@panprog Do you agree with the above explanation by sponsor?

panprog

@nevillehuang @arjun-io The issue is that it's not just a liquidation, it's that the vault will be bricked most of the time while that market's collateral is below margin and above maintenance (due to #23), i.e.:

1. Market's min margin for position is $10. When the weight is 0, market collateral will be set to $10 exactly
2. The following version's price is slightly against the maker, so market's collateral is now $9.999
3. Any vault action will first try to settle (by calling market.update) and since it's below margin, it will revert.

So until the position is liquidated or back above margin - the vault is bricked. It happens both due to this issue and to #23, so this issue is different from #23, but their combination causes this impact.

If the position is back above margin, next vault action will put it back at exactly margin, so the probability of vault bricking is very high and it can be for extended time.

So for all these considerations I still think it's medium.

arjun-io

I see, the confluence with #23 does cause some further issues. Due to the admin's ability to control this weighting I think it's less of an issue but I will defer to the judge

panprog

I believe that sherlock's rules towards admin issues is that if admin does some valid action, but the consequences are unexpected and cause some bad impact (loss of funds / breaking core functionality), then the issue is valid. Here the admin sets weight to 0 in expectation that the market is removed from the vault. Ok, maybe he's aware that collateral will be at min margin and OK with the liquidation fee in such case. But I highly doubt that in such case admin is aware that the vault will be bricked temporarily (which is breaking the core functionality). Note, that in such case admin can not do anything to resume vault operation, because setting weight back to non-0 will revert since it tries to rebalance at the start of this function. That's why I think it's valid medium.

arjun-io

In that case, I agree a medium is appropriate

Issue M-2: Makers can lose funds from price movement even when no long and short positions are opened, due to incorrect distribution of adiabatic fees exposure between makers

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/9

Found by

panprog

Summary

Adiabatic fees introduced in this new update of the protocol (v2.3) were introduced to solve the problem of adiabatic fees netting out to 0 in market token's rather than in USD terms. With the new versions, this problem is solved and adiabatic fees now net out to 0 in USD terms. However, they net out to 0 only for the whole makers pool, but each individual maker can have profit or loss from adiabatic fees at different price levels all else being equal. This creates unexpected risk of loss of funds from adiabatic fees for individual makers, which can be significant, up to several percents of the amount invested.

Vulnerability Detail

The issue is demonstrated in the following scenario:

- price = 1
- Alice open `maker = 10` (`collateral = +0.9` from adiabatic fee)
- Bob opens `maker = 10` (`collateral = +0.7` from adiabatic fee)
- Path A. `price = 1`. Bob closes (final collateral = +0), Alice closes (final collaterral = +0)
- Path B. `price = 2`. Bob closes (final collateral = +0.1), Alice closes (final collaterral = -0.1)
- Path C. `price = 0.5`. Bob closes (final collateral = -0.05), Alice closes (final collateral = +0.05)

Notice that both Alice and Bob are the only makers, there are 0 longs and 0 shorts, but still both Alice and Bob pnl depends on the market price due to pnl from adiabatic fees. Adiabatic fees net out to 0 for all makers aggregated (Alice + Bob),

but not for individual makers. Individual makers pnl from adiabatic fees is more or less random depending on the other makers who have opened.

If Alice were the only maker, then:

- price = 1
- Alice opens `maker = 10` (`collateral = +0.9`)
- price = 2: exposure adjusted +0.9 (Alice `collateral = +1.8`)
- Alice closes `maker = 10` (adiabatic fees = `-1.8`, Alice final collateral = 0)

For the lone maker there is no such problem, final collateral is 0 regardless of price. The core of the issue lies in the fact that the maker's adiabatic fees exposure adjustment is weighted by makers open maker amount. So in the first example:

- price = 1. Alice `maker = 10, exposure = +0.9`, Bob `maker = 10, exposure = +0.7`
- price = 2. Total exposure is adjusted by +1.6, split evenly between Alice and Bob (+0.8 for each)
- Alice new exposure = 0.9 + 0.8 = +1.7 (but adiabatic fees paid to close = -1.8)
- Bob new exposure = 0.7 + 0.8 = +1.5 (but adiabatic fees paid to close = -1.4)

If maker exposure adjustment was weighted by individual makers exposure, then all is correct:

- price = 1. Alice `maker = 10, exposure = +0.9`, Bob `maker = 10, exposure = +0.7`
- price = 2. Total exposure is adjusted by +1.6, split 0.9:0.7 between Alice and Bob, e.g. +0.9 for Alice, +0.7 for Bob
- Alice new exposure = 0.9 + 0.9 = +1.8 (adiabatic fees paid to close = -1.8, net out to 0)
- Bob new exposure = 0.7 + 0.7 = +1.4 (adiabatic fees paid to close = -1.4, net out to 0)

In the worst case, in the example above, if Bob opens `maker = 40` (adiabatic fees `scale = 50`), then at `price = 2`, Alice's final collateral is `-0.4` due to adiabatic fees. Given that Alice's position is 10 at `price = 2` (`notional = 20`), a loss of `-0.4` is a loss of `-2%` at 1x leverage, which is quite significant.

Impact

Individual makers bear an additional undocumented price risk due to adiabatic fees, which is quite significant (can be several percentages of the notional).

Proof of concept

The scenario above is demonstrated in the test, change the following test in test/unit/market/Market.test.ts:

```
it('adiabatic fee', async () => {
  function setupOracle(price: string, timestamp : number, nextTimestamp :
↪  number) {
    const oracleVersion = {
      price: parse6decimal(price),
      timestamp: timestamp,
      valid: true,
    }
    oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
    oracle.status.returns([oracleVersion, nextTimestamp])
    oracle.request.returns()
  }

  async function showInfo() {
    await market.settle(user.address);
    await market.settle(userB.address);
    await market.settle(userC.address);
    var sum : BigNumber = BigNumber.from('0');
    var info = await market.locals(user.address);
    console.log("user collateral = " + info.collateral);
    sum = sum.add(info.collateral);
    var info = await market.locals(userB.address);
    sum = sum.add(info.collateral);
    console.log("userB collateral = " + info.collateral);
    var info = await market.locals(userC.address);
    sum = sum.add(info.collateral);
  }

  async function showVer(ver : number) {
    var v = await market.versions(ver);
    console.log("ver" + ver + ": makerValue=" + v.makerValue + " longValue=" +
↪  v.longValue +
    " makerPosFee=" + v.makerPosFee + " makerNegFee=" + v.makerNegFee +
    " takerPosFee=" + v.takerPosFee + " takerNegFee=" + v.takerNegFee
    );
  }

  const riskParameter = { ...(await market.riskParameter()) }
  const riskParameterMakerFee = { ...riskParameter.makerFee }
  riskParameterMakerFee.linearFee = parse6decimal('0.00')
  riskParameterMakerFee.proportionalFee = parse6decimal('0.00')
  riskParameterMakerFee.adiabaticFee = parse6decimal('0.01')
```

```
  riskParameterMakerFee.scale = parse6decimal('50.0')
  riskParameter.makerFee = riskParameterMakerFee
  const riskParameterTakerFee = { ...riskParameter.takerFee }
  riskParameterTakerFee.linearFee = parse6decimal('0.00')
  riskParameterTakerFee.proportionalFee = parse6decimal('0.00')
  riskParameterTakerFee.adiabaticFee = parse6decimal('0.01')
  riskParameterTakerFee.scale = parse6decimal('50.0')
  riskParameter.takerFee = riskParameterTakerFee
  await market.connect(owner).updateRiskParameter(riskParameter)

  marketParameter = {
    fundingFee: parse6decimal('0.0'),
    interestFee: parse6decimal('0.0'),
    oracleFee: parse6decimal('0.0'),
    riskFee: parse6decimal('0.0'),
    positionFee: parse6decimal('0.0'),
    maxPendingGlobal: 5,
    maxPendingLocal: 3,
    settlementFee: 0,
    makerCloseAlways: false,
    takerCloseAlways: false,
    closed: false,
    settle: false,
  }
  await market.connect(owner).updateParameter(beneficiary.address,
↪  coordinator.address, marketParameter)

  var time = TIMESTAMP;

  setupOracle('1', time, time + 100);
  await market.connect(user)
      ['update(address,uint256,uint256,uint256,int256,bool)'](user.address,
↪  POSITION, 0, 0, COLLATERAL, false);
  await showInfo()
  await showVer(time)

  time += 100;
  setupOracle('1', time, time + 100);
  await market.connect(userB)
      ['update(address,uint256,uint256,uint256,int256,bool)'](userB.address,
↪  POSITION, 0, 0, COLLATERAL, false);
  await showInfo()
  await showVer(time)

  time += 100;
  setupOracle('1', time, time + 100);
  await showInfo()
```


```
  await showVer(time)

  time += 100;
  setupOracle('2', time, time + 100);
  await market.connect(userB)
      ['update(address,uint256,uint256,uint256,int256,bool)'](userB.address, 0,
↪  0, 0, 0, false);
  await showInfo()
  await showVer(time)

  time += 100;
  setupOracle('2', time, time + 100);
  await market.connect(user)
      ['update(address,uint256,uint256,uint256,int256,bool)'](user.address, 0,
↪  0, 0, 0, false);
  await showInfo()
  await showVer(time)

  time += 100;
  setupOracle('0.5', time, time + 100);
  await showInfo()
  await showVer(time)
})
```

Console log:

```
user collateral = 10000000000
userB collateral = 0
ver1636401093: makerValue=0 longValue=0 makerPosFee=0 makerNegFee=0
↪  takerPosFee=0 takerNegFee=0
user collateral = 10000090000
userB collateral = 10000000000
ver1636401193: makerValue=0 longValue=0 makerPosFee=9000 makerNegFee=0
↪  takerPosFee=0 takerNegFee=0
user collateral = 10000090000
userB collateral = 10000070000
ver1636401293: makerValue=0 longValue=0 makerPosFee=7000 makerNegFee=0
↪  takerPosFee=0 takerNegFee=0
user collateral = 10000170000
userB collateral = 10000150000
ver1636401393: makerValue=8000 longValue=0 makerPosFee=0 makerNegFee=0
↪  takerPosFee=0 takerNegFee=0
user collateral = 10000170000
userB collateral = 10000010000
ver1636401493: makerValue=8000 longValue=0 makerPosFee=0 makerNegFee=-14000
↪  takerPosFee=0 takerNegFee=0
```

```
user collateral = 9999990000
userB collateral = 10000010000
ver1636401593: makerValue=-5500 longValue=0 makerPosFee=0 makerNegFee=-4500
↪  takerPosFee=0 takerNegFee=0
```

Notice, that final user balance is -0.1 and final userB balance is +0.1

Code Snippet

Maker exposure is applied to `makerValue`, meaning it's weighted by maker position size: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial/contracts/libs/VersionLib.sol#L314

Tool used

Manual Review

Recommendation

Split the total maker exposure by individual maker's exposure rather than by their position size. To do this:

- Add another accumulator to track total exposure
- Add individual maker `exposure` to user's `Local` storage
- When accumulating local storage in the checkpoint, account global accumulator exposure weighted by individual user's exposure.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/300

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-3: Orders on Optimism chains can not be settled due to revert of `keep()`

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/10

Found by

KingNFT

Summary

After `Ecotone` upgrade, Optimism's `OptGasInfo("0x420000000000000000000000000000000000000F").scalar()` function has been deprecated, which would cause `Kept_Optimism` contract `revert` always.

Reference: https://docs.optimism.io/stack/transactions/fees#l1-data-fee

Vulnerability Detail

The issue arises on L29 of `_calldataFee()` function, as `OPT_GAS.scalar()` would revert.

```
File: contracts\attribute\Kept\Kept_Optimism.sol
14: abstract contract Kept_Optimism is Kept {
...
16:     OptGasInfo constant OPT_GAS =
↪  OptGasInfo(0x420000000000000000000000000000000000000F);

20:     function _calldataFee(
...
24:     ) internal view virtual override returns (UFixed18) {
25:         return _fee(
26:             OPT_GAS.getL1GasUsed(applicableCalldata),
27:             multiplierCalldata,
28:             bufferCalldata,
29:             OPT_GAS.l1BaseFee() * OPT_GAS.scalar() / (10 **
↪  OPT_GAS.decimals()) // @audit revert due to OPT_GAS.scalar()
30:         );
31:     }
32: }
```

The following PoC is built on both Optimism and Base mainnet, we can see `OPT_GAS.scalar()` reverts with `GasPriceOracle: scalar() is deprecated` message.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

import "forge-std/Test.sol";

interface OptGasInfo {
    function scalar() external view returns (uint256);
}

contract OptimismKeepFeeBug is Test {
    uint256 constant BLOCK_HEIGHT = 12520000;//118120000; // Mar-30-2024
↪   10:46:17 PM +UTC
    string constant RPC_URL =
↪   "https://mainnet.base.org";//"https://mainnet.optimism.io";
    OptGasInfo constant OPT_GAS =
↪   OptGasInfo(0x420000000000000000000000000000000000000F);

    function setUp() public {
        vm.createSelectFork(RPC_URL, BLOCK_HEIGHT);
    }

    function testOptGasInfoScalarCallRevert() public {
        vm.expectRevert();
        OPT_GAS.scalar();
    }
}
```

the test log:

```
2024-02-perennial-v2-3\root> forge test --mc OptimismKeepFeeBug -vvvv
[] Compiling...
[] Compiling 1 files with 0.8.23Compiler run successful!
[] Compiling 1 files with 0.8.23
[] Solc 0.8.23 finished in 2.48s

Running 1 test for test/OptimismKeepFeeBug.t.sol:OptimismKeepFeeBug
[PASS] testOptGasInfoScalarCallRevert() (gas: 13337)
Traces:
  [13337] OptimismKeepFeeBug::testOptGasInfoScalarCallRevert()
    [0] VM::expectRevert(custom error f4844814:)
        ()
    [7434] 0x420000000000000000000000000000000000000F::scalar() [staticcall]
      [2436] 0xb528D11cC114E026F138fE568744c6D45ce6Da7A::scalar()
↪   [delegatecall]
            revert: GasPriceOracle: scalar() is deprecated
```

```
          revert: GasPriceOracle: scalar() is deprecated
      ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 842.07ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Impact

Orders can not be settled, break of core functionality.

Code Snippet

https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/root/contracts/attribute/Kept/Kept_Optimism.sol#L29

Tool used

Manual Review

Recommendation

reference: https://docs.optimism.io/stack/transactions/fees#ecotone

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/root/pull/90

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-4: All transactions to claim assets from the vault will revert in some situations due to double subtraction of the claimed assets in market position allocations calculation.

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/11

Found by

panprog

Summary

When assets are claimed from the vault (`Vault.update(0,0,x)` called), the vault rebalances its collateral. There is an issue with market positions allocation calculations: the assets ("total position") subtract claimed amount twice. This leads to revert in case this incorrect `assets` amount is less than `minAssets` (caused by market's `minPosition`). In situations when the vault can't redeem due to some market's position being at the `minPosition` (because of the market's skew, which disallows makers to reduce their positions), this will lead to all users being unable to claim any assets which were already redeemed and settled.

Vulnerability Detail

`Vault.update` rebalances collateral by calling `_manage`:

```
_manage(context, depositAssets, claimAmount, !depositAssets.isZero() ||
↪  !redeemShares.isZero());
```

In the rebalance calculations, collateral and assets (assets here stands for "total vault position") are calculated as following:

```
UFixed6 collateral = UFixed6Lib.unsafeFrom(strategy.totalCollateral).add(deposit
↪  ).unsafeSub(withdrawal);
UFixed6 assets = collateral.unsafeSub(ineligable);

if (collateral.lt(strategy.totalMargin)) revert
↪  StrategyLibInsufficientCollateralError();
if (assets.lt(strategy.minAssets)) revert StrategyLibInsufficientAssetsError();
```

`ineligable` is calculated as following:

```
function _ineligable(Context memory context, UFixed6 withdrawal) private pure
↪  returns (UFixed6) {
    // assets eligable for redemption
    UFixed6 redemptionEligable = UFixed6Lib.unsafeFrom(context.totalCollateral)
        .unsafeSub(withdrawal)
        .unsafeSub(context.global.assets)
        .unsafeSub(context.global.deposit);

    return redemptionEligable
        // approximate assets up for redemption
        .mul(context.global.redemption.unsafeDiv(context.global.shares.add(conte
↪  xt.global.redemption)))
        // assets pending claim
        .add(context.global.assets)
        // assets withdrawing
        .add(withdrawal);
}
```

Notice that `ineligable` adds `withdrawal` in the end (which is the assets claimed by the user). Now back to collateral and assets calculation:

- `collateral = totalCollateral + deposit - withdrawal`
- `assets = collateral - ineligable = collateral - (redemptionEligable * redemption / (redemption + shares) + global.assets + withdrawal)`
- `assets = totalCollateral + deposit - withdrawal - [redemptionIneligable] - global.assets - withdrawal`
- `assets = totalCollateral + deposit - [redemptionIneligable] - global.assets - 2 * withdrawal`

See that `withdrawal` (assets claimed by the user) is subtracted twice in assets calculations. This means that assets calculated are smaller than it should. In particular, assets might become less than minAssets thus reverting in the following line:

```
if (assets.lt(strategy.minAssets)) revert StrategyLibInsufficientAssetsError();
```

Possible scenario for this issue to cause inability to claim funds:

1. Some vault market's has a high skew (|long - short|), which means that minimum maker position is limited by the skew.
2. User redeems large amount from the vault, reducing vault's position in that market so that market maker ~= |long - short|. This means that further redeems from the vault are not possible because the vault can't reduce its

position in the market.

3. After that, the user tries to claim what he has redeemed, but all attempts to redeem will revert (both for this user and for any other user that might want to claim)

Impact

In certain situations (redeem not possible from the vault due to high skew in some underlying market) claiming assets from the vault will revert for all users, temporarily (and sometimes permanently) locking user funds in the contract.

Proof of concept

The scenario above is demonstrated in the test, change the following test in test/integration/vault/Vault.test.ts:

```
    it('simple deposits and redemptions', async () => {
...
      // Now we should have opened positions.
      // The positions should be equal to (smallDeposit + largeDeposit) *
↪  leverage originalOraclePrice.
      expect(await position()).to.equal(
        smallDeposit.add(largeDeposit).mul(leverage).mul(4).div(5).div(originalO
↪  raclePrice),
      )
      expect(await btcPosition()).to.equal(
        smallDeposit.add(largeDeposit).mul(leverage).div(5).div(btcOriginalOracl
↪  ePrice),
      )

      /*** remove all lines after this and replace with the following code: ***/

      var half = smallDeposit.add(largeDeposit).div(2).add(smallDeposit);
      await vault.connect(user).update(user.address, 0, half, 0)

      await updateOracle()
      await vault.connect(user2).update(user2.address, smallDeposit, 0, 0) //
↪  this will create min position in the market
      await vault.connect(user).update(user.address, 0, 0, half) // this will
↪  revert even though it's just claiming
    })
```

The last line in the test will revert, even though it's just claiming assets. If the pre-last line is commented out (no "min position" created in the market), it will work normally.

Code Snippet

Ineligable amount calculation adds `withdrawal`:
https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L431

`withdrawal` is subtracted twice - once directly from collateral, 2nd time via ineligable amount subtractions:
https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L118-L119

Tool used

Manual Review

Recommendation

Remove `add(withdrawal)` from `_ineligable` calculation in the vault.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/303

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-5: If referral or liquidator is the same address as the account, then liquidation/referral fees will be lost due to local storage being overwritten after the `claimable` amount is credited to liquidator or referral

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/16

Found by

bin2chen, panprog

Summary

Any user (address) can be liquidator and/or referral, including account's own address (the user can self-liquidate or self-refer). During the market settlement, liquidator and referral fees are credited to liquidator/referral's `local.claimable` storage. The issue is that the account's local storage is held in the memory during the settlement process, and is saved into storage after settlement/update. This means that `local.claimable` storage changes for the account are not reflected in the in-memory cached copy and discarded when the cached copy is saved after settlement.

This leads to liquidator and referral fees being lost when these are the account's own address.

Vulnerability Detail

During market account settlement process, in the `_processOrderLocal`, liquidator and referral fees are credited to corresponding accounts via:

```
...
    _credit(liquidators[account][newOrderId], accumulationResult.liquidationFee);
    _credit(referrers[account][newOrderId], accumulationResult.subtractiveFee);
...
function _credit(address account, UFixed6 amount) private {
    if (amount.isZero()) return;

    Local memory newLocal = _locals[account].read();
    newLocal.credit(amount);
    _locals[account].store(newLocal);
}
```

However, for the account the cached copy of `_locals[account]` is stored after the settlement in `_storeContext`:

```
function _storeContext(Context memory context, address account) private {
    // state
    _global.store(context.global);
    _locals[account].store(context.local);
...
```

The order of these actions is:

```
function settle(address account) external nonReentrant whenNotPaused {
    Context memory context = _loadContext(account);

    _settle(context, account);

    _storeContext(context, account);
}
```

1. Load `_locals[account]` into memory (`context.local`)
2. Settle: during settlement `_locals[account].claimable` is increased for liquidator and referral. Note: this is not reflected in `context.local`
3. Store cached context: `_locals[account]` is overwritten with the `context.local`, losing `claimable` increased during settlement.

Impact

If user self-liquidates or self-refers, the liquidation and referral fees are lost by the user (and are stuck in the contract, because they're still subtracted from the user's collateral).

Proof of concept

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

```
it('self-liquidation fees lost', async () => {
const POSITION = parse6decimal('100.000')
const COLLATERAL = parse6decimal('120')

function setupOracle(price: string, timestamp : number, nextTimestamp : number) {
    const oracleVersion = {
    price: parse6decimal(price),
    timestamp: timestamp,
    valid: true,
```


```
    }
    oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
    oracle.status.returns([oracleVersion, nextTimestamp])
    oracle.request.returns()
}

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

var time = TIMESTAMP;

setupOracle('1', time, time + 100);
await market.connect(user)
    ['update(address,uint256,uint256,uint256,int256,bool)'](user.address,
↪  POSITION, 0, 0, COLLATERAL, false);

time += 100;
setupOracle('1', time, time + 100);
await market.connect(userB)
    ['update(address,uint256,uint256,uint256,int256,bool)'](userB.address, 0,
↪  POSITION, 0, COLLATERAL, false);

time += 100;
setupOracle('1', time, time + 100);

time += 100;
setupOracle('0.7', time, time + 100);

// self-liquidate
setupOracle('0.7', time, time + 100);
await market.connect(userB)
    ['update(address,uint256,uint256,uint256,int256,bool)'](userB.address, 0, 0,
↪  0, 0, true);

// settle liquidation
time += 100;
setupOracle('0.7', time, time + 100);
await market.settle(userB.address);
var info = await market.locals(userB.address);
console.log("Claimable userB: " + info.claimable);
```

Console log:

```
Claimable userB: 0
```


Code Snippet

`Market._credit` modifies `local.claimable` storage for the account: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L678-L684

Tool used

Manual Review

Recommendation

Modify `Market._credit` function to increase `context.local.claimable` if account to be credited matches account which is being updated.

Discussion

sherlock-admin2

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

takarez commented:

> this seems valid valid medium; medium(2)

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/equilibria-xyz/perennial-v2/pull/302

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-6: _loadContext() uses the wrong pendingGlobal.

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/17

Found by

bin2chen

Summary

`StrategyLib._loadContext()` is using the incorrect `pendingGlobal`, causing `currentPosition`, `minPosition`, and `maxPosition` to be incorrect, leading to incorrect rebalance operation.

Vulnerability Detail

In `StrategyLib._loadContext()`, there is a need to compute `currentPosition`, `minPosition`, and `maxPosition`. The code as follows:

```
    function _loadContext(
        Registration memory registration
    ) private view returns (MarketStrategyContext memory marketContext) {
...
        // current position
@>      Order memory pendingGlobal = registration.market.pendings(address(this));
        marketContext.currentPosition = registration.market.position();
        marketContext.currentPosition.update(pendingGlobal);
        marketContext.minPosition = marketContext.currentAccountPosition.maker
            .unsafeSub(marketContext.currentPosition.maker
                .unsafeSub(marketContext.currentPosition.skew().abs()).min(marke
↪   tContext.closable));
        marketContext.maxPosition = marketContext.currentAccountPosition.maker
            .add(marketContext.riskParameter.makerLimit.unsafeSub(marketContext.
↪   currentPosition.maker));
    }
```

The code above `pendingGlobal = registration.market.pendings(address(this));` is wrong It takes the address(this)'s `pendingLocal`. The correct approach is to use `pendingGlobal = registration.market.pending();`.

Impact

Since `pendingGlobal` is wrong, `currentPosition`, `minPosition` and `maxPosition` are all wrong. affects subsequent rebalance calculations, such as `target.position` etc.

rebalance does not work properly

Code Snippet

https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L200

Tool used

Manual Review

Recommendation

```
    function _loadContext(
        Registration memory registration
    ) private view returns (MarketStrategyContext memory marketContext) {
...
        // current position
-       Order memory pendingGlobal = registration.market.pendings(address(this));
+       Order memory pendingGlobal = registration.market.pending();
        marketContext.currentPosition = registration.market.position();
        marketContext.currentPosition.update(pendingGlobal);
        marketContext.minPosition = marketContext.currentAccountPosition.maker
            .unsafeSub(marketContext.currentPosition.maker
                .unsafeSub(marketContext.currentPosition.skew().abs()).min(marke
↪   tContext.closable));
        marketContext.maxPosition = marketContext.currentAccountPosition.maker
            .add(marketContext.riskParameter.makerLimit.unsafeSub(marketContext.
↪   currentPosition.maker));
    }
```

Discussion

sherlock-admin4

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

panprog commented:

> valid medium, it influences the rebalance process only in very rare edge cases

takarez commented:

> the reason for it should have been said.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/299

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-7: Liquidator can set up referrals for other users

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/22

Found by

bin2chen

Summary

If a user has met the liquidation criteria and currently has no referrer then a malicious liquidator can specify a referrer in the liquidation order. making it impossible for subsequent users to set up the referrer they want.

Vulnerability Detail

Currently, there are 2 conditions to set up a referrer

1. the order cannot be empty Non-empty orders require authorization unless they are liquidation orders
2. there can't be another referrer already

```
    function _loadUpdateContext(
        Context memory context,
        address account,
        address referrer
    ) private view returns (UpdateContext memory updateContext) {
...
        updateContext.referrer = referrers[account][context.local.currentId];
        updateContext.referralFee =
↪   IMarketFactory(address(factory())).referralFee(referrer);
    }

    function _processReferrer(
        UpdateContext memory updateContext,
        Order memory newOrder,
        address referrer
    ) private pure {
@>      if (newOrder.makerReferral.isZero() && newOrder.takerReferral.isZero())
↪   return;
        if (updateContext.referrer == address(0)) updateContext.referrer =
↪   referrer;
        if (updateContext.referrer == referrer) return;
```

```
        revert MarketInvalidReferrerError();
    }


    function _storeUpdateContext(Context memory context, UpdateContext memory
↪   updateContext, address account) private {
...
        referrers[account][context.local.currentId] = updateContext.referrer;
    }
```

However, if the user does not have a referrer, the liquidation order is able to meet both of these restrictions

This allows the liquidator to set up referrals for other users.

When the user subsequently tries to set up a referrer, it will fail.

Impact

If a user is set up as a referrer by a liquidated order in advance, the user cannot be set up as anyone else.

Code Snippet

https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L503

Tool used

Manual Review

Recommendation

Restrictions on Liquidation Orders Cannot Set a referrer

```
    function _processReferrer(
        UpdateContext memory updateContext,
        Order memory newOrder,
        address referrer
    ) private pure {
+       if (newOrder.protected() && referrer != address(0)) revert
↪   MarketInvalidReferrerError;
        if (newOrder.makerReferral.isZero() && newOrder.takerReferral.isZero())
↪   return;
        if (updateContext.referrer == address(0)) updateContext.referrer =
↪   referrer;
```

```
        if (updateContext.referrer == referrer) return;

        revert MarketInvalidReferrerError();
    }
```

Discussion

sherlock-admin3

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

panprog commented:

> invalid, because this is the system design: referral can receive some fee from the user, and when liquidator liquidates - his (liquidator's) referral will get some fee from the user. Moreover, the referral is set for each order, and since no order can be placed until liquidation settles, the user can't execute orders for the same oracle version anyway, and for the following oracle versions a new referrer can easily be set

nevillehuang

@bin2chen66 This seems invalid based on @panprog comment.

bin2chen66

The system design doesn't seem to be like this. Could it be that I misunderstood? The referrer is set and cannot be modified.

1. _loadUpdateContext () take the previous referrer
2. context.local.currentId unchanged or + 1
3. _processReferrer () Verify that updateContext.referrer must be equal to the referrer of the current order
4. _storeUpdateContext () save referrers [account] [context.local.currentId]

The referrer inherits the previous one and verifies that it cannot be reset.
@Panprog Can you see where can reset it, Did I miss it? Thanks.

panprog

@bin2chen66 @nevillehuang Yes, I agree with @bin2chen66 , the referrer can not be modified. Sorry for incorrect comment, I've missed that it's not reset between updates. I believe this is medium then as it allows to set the referral for the user during liquidation which he can't change.

arjun-io

It's accurate that the liquidator can set a referral address as part of the liquidation - this is acceptable. The referrer address is locked for the current ID but future orders (for a different local.currentId) should not have the referrer set - if they do that would be a bug. So as long as the liquidation version is filled, new orders for the next version should be fine.

nevillehuang

@arjun-io So do you agree this is a valid issue? I am incline to keep medium given referrer setting can be blocked

arjun-io

No I don't think it is, the only way this would be valid is if setting referrer is blocked for *future* orders that are not from the same Oracle Version. Would defer to auditors to double check this

panprog

@nevillehuang @arjun-io Yes, after the referrer is set once, user can never change it again, because it's loaded from currentId into updateContext, but then if `local.currentId` increases, the same referrer (from previous currentId) is stored into new currentId, thus referrer is always carried over from previous ids. So it seems that the issue which is valid is that it's impossible to change referrer once set, not that liquidator can set his referrer, although it wasn't really obvious from the docs (the intended functionality of setting the referrer).

arjun-io

Ah I see, the issue arises from the _storeUpdateContext using an updated context.local.currentId - this would also be an issue for liquidations then, I believe. In which case this is a deeper issue which might be a High

panprog

@arjun-io Do you mean that liquidator carries over from previous ids to currentId? Yes, it carries over like referrer, however, the accumulated `liquidationFee` will be 0 for all orders which are not protected (`CheckPointLib._accumulateLiquidationFee`):

```
function _accumulateLiquidationFee(
    Order memory order,
    Version memory toVersion
) private pure returns (UFixed6 liquidationFee) {
    if (order.protected())
        return
↪   toVersion.liquidationFee.accumulated(Accumulator6(Fixed6Lib.ZERO),
↪   UFixed6Lib.ONE).abs();
}
```

So there is no issue with liquidators. Even though the liquidators map will be set for all currentIds, this liquidator will be credited only once during the liquidation, and in following liquidations it will be overwritten with new liquidator. Yes, it's better to fix it so that liquidator doesn't carry over from previoud ids, but there is no impact in this right now.

And the impact for the issue described here I think is medium, not high.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/297

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-8: Vault and oracle keepers DoS in some situations due to `market.update(account,max,max,max,0,false)`

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/23

Found by

panprog

Summary

When user's market account is updated without position and collateral change (by calling `market.update(account,max,max,max,0,false)`), this serves as some kind of "settling" the account (which was the only way to settle the account before v2.3). However, this action still reverts if the account is below margin requirement.

The issue is that some parts of the code use this action to "settle" the account in the assumption that it never reverts which is not true. This causes unpexpected reverts and denial of service to users who can not execute transactions in some situations, in particular:

1. Oracle `KeeperFactory.settle` uses this method to settle all accounts in the market for the oracle verison and will revert entire market version's settlement if any account which is being settled is below margin requirement. Example scenario: 1.1. User increases position to the edge of margin requirement 1.2. The price rises slightly for the commited oracle version, and user position is settled and is now slightly below margin requirements 1.3. All attempts to settle accounts for the commited oracle version for this market will revert as user's account collateral is below margin requirements.
2. Vault `Vault._updateUnderlying` uses this method to settle all vault's accounts in the markets. This function is called at the start of `rebalance` and `update`, with `rebalance` also being called before any admin vault parameters changes such as updating market leverages, weights or cap. This becomes especially problematic if any market is "removed" from the vault by setting its weight to 0, but the market still has some position due to `minPosition` limitation (as described in another issue). In such case each vault update will bring this market's position to exact edge of margin requirement, meaning a lot of times minimal price changes will put the vault's market account below margin requirement, and as such most Vault functions will revert (`update`, `rebalance` and admin param changes). Moreover, since the vault rebalances collateral and/or position size only in `_manage` (which is called only from `update` and `rebalance`), this means that the vault is basically bricked until this position is

either liquidated or goes above margin requirement again due to price changes.

Vulnerability Detail

When `Market.update` is called, any parameters except `protected = true` will perform the following check from the `InvariantLib.validate`:

```
if (
    !PositionLib.margined(

↪   context.latestPosition.local.magnitude().add(context.pending.local.pos()),
        context.latestOracleVersion,
        context.riskParameter,
        context.local.collateral
    )
) revert IMarket.MarketInsufficientMarginError();
```

This means that even updates which do not change anything (empty order and 0 collateral change) still perform this check and revert if the user's collateral is below margin requirement.

Such method to settle accounts is used in `KeeperOracle._settle`:

```
function _settle(IMarket market, address account) private {
    market.update(account, UFixed6Lib.MAX, UFixed6Lib.MAX, UFixed6Lib.MAX,
↪   Fixed6Lib.ZERO, false);
}
```

This is called from `KeeperFactory.settle`, which the keepers are supposed to call to settle market accounts after the oracle version is commited. This will revert, thus keepers will temporarily be unable to call this function for the specific oracle version until all users are at or above margin.

The same method is used to settle accounts in `Vault._updateUnderlying`:

```
function _updateUnderlying() private {
    for (uint256 marketId; marketId < totalMarkets; marketId++)
        _registrations[marketId].read().market.update(
            address(this),
            UFixed6Lib.MAX,
            UFixed6Lib.ZERO,
            UFixed6Lib.ZERO,
            Fixed6Lib.ZERO,
            false
        );
```

```
}
```

Impact

1. Keepers are unable to settle market accounts for the commited oracle version until all accounts are above margin. The oracle fees are still taken from all accounts, but the keepers are blocked from receiving it.
2. If any Vault's market weight is set to 0 (or if vault's position in any market goes below margin for whatever other reason), most of the time the vault will temporarily be bricked until vault's position in that market is liquidated. The only function working in this state is `Vault.settle`, even all admin functions will revert.

Code Snippet

`InvariantLib.validate` reverts for all updates (except liquidations) where account is below margin requirements:
https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial/contracts/libs/InvariantLib.sol#L78-L85

`KeeperOracle._settle` uses `Market.update` to settle accounts:
https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-oracle/contracts/keeper/KeeperOracle.sol#L178-L180

`Vault._updateUnderlying` also uses the same method to settle accounts:
https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L342-L352

Tool used

Manual Review

Recommendation

Depending on intended functionality:

1. Ignore the margin requirement for empty orders and collateral change which is >= 0. **AND/OR**
2. Use `Market.settle` instead of `Market.update` to settle accounts, specifically in `KeeperOracle._settle` and in `Vault._updateUnderlying`. There doesn't seem to be any reason or issue to use `settle` instead of `update`, it seems that `update` is there just because there was no `settle` function available before.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/309

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-9: Vault checkpoints slightly incorrect conversion from assets to shares leads to slow loss of funds for long-time vault depositors

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/25

Found by

panprog

Summary

When vault checkpoints convert assets to shares (specifically used to calculate user's shares for their deposit), it uses the following formula: `shares = (assets[before fee] - settlementFee) * checkpoint.shares/checkpoint.assets * (deposit + redeem - tradeFee) / (deposit + redeem)`

`settlementFee` in this formula is taken into account slightly incorrectly: in actual market collateral calculations, both settlement fee and trade fee are subtracted from collateral, but this formula basically multiplies `1 - settlement fee percentage` by `1 - trade fee percentage`, which is slightly different and adds the calculation error = `settlement fee percentage * trade fee percentage`.

This is the scenario to better understand the issue:

1. Linear fee = 2%, settlement fee = $1
2. User1 deposits $100 into the vault (linear fee = $2, settlement fee = $1)
3. Vault assets = $97 (due to fees), User1 shares = 100
4. User2 deposits $100 into the vault (linear fee = $2, settlement fee = $1)
5. Vault assets = $194, User1 shares = 100, but User2 shares = 100.02, meaning User1's share value has slightly fallen due to a later deposit.

This is the calculation for User2 shares: `shares = ($100 - $1) * 100/$97 * ($100 - $2) / $100 = $99 * 100/$97 * $98/$100 = $99 * 98/$97 = 100.02`

The extra 0.02 this user has received is because the `tradeFee` is taken from the amount after settlement fee ($99) rather than full amount as it should (100). This difference (`settlementFee * tradeFee = $0.02`) is unfair amount earned by User2 and loss of funds for User1.

When redeeming, the formula for shares -> assets vault checkpoint conversion is correct and the correct amount is redeemed.

This issue leads to all vault depositors slowly losing share value with each deposit, and since no value is gained when redeeming, continuous deposits and redeems will lead to all long-time depositors continuously losing their funds.

Vulnerability Detail

This is the formula for vault checkpoint `toSharesGlobal`:

```
function toSharesGlobal(Checkpoint memory self, UFixed6 assets) internal pure
↪   returns (UFixed6) {
    // vault is fresh, use par value
    if (self.shares.isZero()) return assets;

    // if vault is insolvent, default to par value
    return  self.assets.lte(Fixed6Lib.ZERO) ? assets : _toShares(self,
↪   _withoutSettlementFeeGlobal(self, assets));
}

function _toShares(Checkpoint memory self, UFixed6 assets) private pure returns
↪   (UFixed6) {
    UFixed6 selfAssets = UFixed6Lib.unsafeFrom(self.assets);
    return _withSpread(self, assets.muldiv(self.shares, selfAssets));
}

function _withSpread(Checkpoint memory self, UFixed6 amount) private pure
↪   returns (UFixed6) {
    UFixed6 selfAssets = UFixed6Lib.unsafeFrom(self.assets);
    UFixed6 totalAmount = self.deposit.add(self.redemption.muldiv(selfAssets,
↪   self.shares));
    UFixed6 totalAmountIncludingFee =
↪   UFixed6Lib.unsafeFrom(Fixed6Lib.from(totalAmount).sub(self.tradeFee));

    return totalAmount.isZero() ?
        amount :
        amount.muldiv(totalAmountIncludingFee, totalAmount);
}

function _withoutSettlementFeeGlobal(Checkpoint memory self, UFixed6 amount)
↪   private pure returns (UFixed6) {
    return _withoutSettlementFee(amount, self.settlementFee);
}

function _withoutSettlementFee(UFixed6 amount, UFixed6 settlementFee) private
↪   pure returns (UFixed6) {
    return amount.unsafeSub(settlementFee);
}
```

This code translates to a formula shown above, i.e. it first subtracts settlement fee from the assets (`withoutSettlementFeeGlobal`), then multiplies this by checkpoint's share value in `_toShares` (`*checkpoint.shares/checkpoint.assets`), and then multiplies this by trade fee adjustment in `_withSpread` (`*(deposit+redeem-tradeFee) / (deposit+redeem)`). Here is the formula again: `shares = (assets[before fee] - settlementFee) * checkpoint.shares/checkpoint.assets * (deposit + redeem - tradeFee) / (deposit + redeem)`

As shown above, the formula is incorrect, because it basically does the following: `user_assets = (deposit - settlementFee) * (deposit - tradeFee)/deposit = deposit * (1 - settlementFeePct) * (1 - tradeFeePct)`

But the actual user collateral after fees is calculated as: `user_assets = deposit - settlementFee - tradeFee = deposit * (1 - settlementFeePct - tradeFeePct)`

If we subtract the actual collateral from the formula used in checkpoint, we get the error: `error = deposit * ((1 - settlementFeePct) * (1 - tradeFeePct) - (1 - settlementFeePct - tradeFeePct)) error = deposit * settlementFeePct * tradeFeePct error = settlementFee * tradeFeePct`

So this is systematic error, which inflates the shares given to users with any deposit by fixed amount of `settlementFee * tradeFeePct`

Impact

Any vault deposit reduces the vault assets by `settlementFee * tradeFeePct`. While this amount is not very large (in the order of $0.1 - $0.001 per deposit transaction), this is amount lost with each deposit, and given that an active vault can easily have 1000s of transactions daily, this will be a loss of $1–100/day, which is significant enough to make it a valid issue.

Code Snippet

SettlementFee subtracted from asset before proceeding in `toSharesGlobal`: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/types/Checkpoint.sol#L91-L97

The result is multiplied by the checkpoint's share to assets ratio in `_toShares`: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/types/Checkpoint.sol#L153-L156

And the final result is multiplied by `tradeFee`-adjusted deposits and redeems in `_withSpread`: https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-vault/contracts/types/Checkpoint.sol#L169-L177

Tool used

Manual Review

Recommendation

Re-work the assets to shares conversion in vault checkpoint to use the correct formula: `shares = (assets[before fee] - settlementFee - tradeFee * assets / (deposit + redeem)) * checkpoint.shares/checkpoint.assets`

Discussion

sherlock-admin2

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

takarez commented:

> this seem valid medium; medium(3)

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/304

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-10: ChainlinkFactory will pay non-requested versions keeper fees

Source: https://github.com/sherlock-audit/2024-02-perennial-v2-3-judging/issues/32

Found by

bin2chen

Summary

Protocol definition: `Requested versions will pay out a keeper fee, non-requested versions will not.` But ChainlinkFactory ignores `numRequested`, which pays for both.

Vulnerability Details

Protocol definition: `Requested versions will pay out a keeper fee, non-requested versions will not.`

```
/// @notice Commits the price to specified version
/// @dev Accepts both requested and non-requested versions.
///      Requested versions will pay out a keeper fee, non-requested versions
↪  will not.
///      Accepts any publish time in the underlying price message, as long as it
↪  is within the validity window,
///      which means its possible for publish times to be slightly out of order
↪  with respect to versions.
///      Batched updates are supported by passing in a list of price feed ids
↪  along with a valid batch update data.
/// @param ids The list of price feed ids to commit
/// @param version The oracle version to commit
/// @param data The update data to commit
function commit(bytes32[] memory ids, uint256 version, bytes calldata data)
↪  external payable {
```

`commit()->_handleKeeperFee()->_applicableValue()`
`ChainlinkFactory._applicableValue ()` implements the following:

```
function _applicableValue(uint256, bytes memory data) internal view override
↪  returns (uint256) {
    bytes[] memory payloads = abi.decode(data, (bytes[]));
    uint256 totalFeeAmount = 0;
    for (uint256 i = 0; i < payloads.length; i++) {
```

```
        (, bytes memory report) = abi.decode(payloads[i], (bytes32[3], bytes));
        (Asset memory fee, ,) = feeManager.getFeeAndReward(address(this),
↪   report, feeTokenAddress);
        totalFeeAmount += fee.amount;
    }
    return totalFeeAmount;
}
```

The above method ignores the first parameter `numRequested`. This way, whether it is `Requested versions` or not, you will pay `keeper fees`. Violating `non-requested versions will not pay`

Impact

If `non-requested versions` will pay as well, it is easy to maliciously submit `non-requested` maliciously consume `ChainlinkFactory` fees balance (Note that needs at least one numRequested to call `_handleKeeperFee()`)

Code Snippet

https://github.com/sherlock-audit/2024-02-perennial-v2-3/blob/main/perennial-v2/packages/perennial-oracle/contracts/chainlink/ChainlinkFactory.sol#L71

Tool used

Manual Review

Recommendation

It is recommended that only `Requested versions` keeper fees'

```
-   function _applicableValue(uint256 , bytes memory data) internal view
↪   override returns (uint256) {
+   function _applicableValue(uint256 numRequested, bytes memory data) internal
↪   view override returns (uint256) {
        bytes[] memory payloads = abi.decode(data, (bytes[]));
        uint256 totalFeeAmount = 0;
        for (uint256 i = 0; i < payloads.length; i++) {
            (, bytes memory report) = abi.decode(payloads[i], (bytes32[3],
↪   bytes));
            (Asset memory fee, ,) = feeManager.getFeeAndReward(address(this),
↪   report, feeTokenAddress);
            totalFeeAmount += fee.amount;
        }
-       return totalFeeAmount;
```

```
+       return totalFeeAmount * numRequested / payloads.length ;
    }
```

Discussion

sherlock-admin2

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

panprog commented:

> valid medium, the attacker will have to commit requested along with unrequested which might not be easy to do due to competition

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/293

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Disclaimers

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

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