CANTINA

Desk-Orderbook

Competition

February 17, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Desk-Orderbook is an innovative pool-based perpetual DEX protocol designed to offer a range of advanced features. It introduces multi-asset collateral support and cross-margin flexibility, providing traders with enhanced options and opportunities.

The protocol incorporates secured measurements, including virtual price impact and funding fees, to ensure the protection of liquidity providers (LPs) from being overly exposed to a single direction. By implementing these measures, Desk-Orderbook aims to create a more resilient and balanced trading environment.

From Jan 6th to Jan 20th Cantina hosted a competition based on desk-orderbook. The participants identified a total of **25** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 5
- Low Risk: 12
- Gas Optimizations: 0
- Informational: 7

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Incorrect Margin Calculation Due to Inconsistent Realized and Unrealized Balances

Submitted by Tripathi

Severity: High Risk

Context: LiquidationHandler.sol#L77

Summary: The protocol supports multi-collateral and uses collateral factors to calculate the minimum margin required for liquidation. Due to inconsistent treatment of realized and unrealized settlement token balances in the margin calculation, users may be incorrectly liquidated.

Vulnerability Details: The liquidation handler calculates `_totalMarginWithUnsettledE18` to determine if an account should be liquidated by comparing it against the Minimum Margin Requirement (MMR). The current implementation treats realized and unrealized settlement token balances differently when applying collateral factors:

```
function _executeAction(bytes memory _data) internal override {
    // ...

    int256 _borrowingFeeE18 = _assetService.getSubaccountPendingBorrowingFee(_action.subaccount);
    (int256 _totalUPnLE18, int256 _totalFundingFeeE18) =
        _perpService.getSubaccountTotalUnrealizedPNLAndFundingFee(_action.subaccount);

    int256 _unsettledE18 = _totalUPnLE18 - _totalFundingFeeE18 - _borrowingFeeE18;

    // apply collateral factor if unsettled balance is positive
    if (_unsettledE18 > 0) {
        _unsettledE18 =
            _unsettledE18 * _assetService.collateralFactors(_settlementToken).toInt256() / int256(BPS);
    }
    _totalMarginWithUnsettledE18 = _assetService.getSubaccountTotalMargin(_action.subaccount) + _unsettledE18;

    // ...
```

This is from liquidationHandler module, where we can see the calculation of the `_totalMarginWithUnsettledE18` which is check against MMR to decide weather the account should be liquidated or not. Users margin will change if he moves unrealised balance to realised vice versa but ideally margin should be independent of realised and unrealised balance. Margin only depend on the total debt and supplied collateral.

The issue arises because the implementation:

1. Applies collateral factors separately to realized and unrealized balances for settlement token.
2. Creates discrepancies in margin calculations between two identical positions that only differ in their realized/unrealized ratio.

For example:

- User A: -500 USDC unrealized + 600 USDC realized.
- User B: -400 USDC unrealized + 500 USDC realized (after realizing 100 USDC loss).

Ideallly both should be equal collateral but it isn't true.

Impact Explanation: Incorrect liquidation.

Proof of Concept: The following test demonstrates how moving balances between realized and unrealized states changes the margin:

```
function test_IncorrectMarginCalculation() public {
    vm.prank(DEPLOYER);
   // Set USDC collateral factor to 0.9 (9000 basis points)
    assetService.setCollateralFactor(address(USDC), 9000);

    // Scenario 1: Alice with -500 USDC unrealized and +600 USDC realized
    // Set up Alice's initial state
    vm.prank(WHITELIST);
```

```
    assetService.adjustCollateral(aliceSubId, address(USDC), int256(600 * 1e6));  // Realized +600 USDC
    // unrealized pnl = -500 USDC, funding fee = 0 (for simplicity)
    vm.mockCall(
            address(perpService),
            abi.encodeWithSelector(PerpService.getSubaccountTotalUnrealizedPNLAndFundingFee.selector,
            ↪  aliceSubId),
            abi.encode(-int256(500 ether), 0)
    );

   (int256 totalUPNL1, ) = perpService.getSubaccountTotalUnrealizedPNLAndFundingFee(aliceSubId);
   int256 collateral1 = assetService.getSubaccountTotalMargin(aliceSubId) + totalUPNL1; // USDC

    // Calculate margin for Scenario 1
    // Realized: +600 USDC * 0.9 (collat factor) = +540 USDC
    // Unrealized: -500 USDC (no collat factor for negative values)
    // Total: 540 - 500 = 40 USDC

    // Scenario 2: Move 100 USDC unrealized loss to realized
    vm.prank(WHITELIST);
    assetService.adjustCollateral(aliceSubId, address(USDC), -int256(100 * 1e6));  // Realized now +500 USDC

    vm.mockCall(
            address(perpService),
            abi.encodeWithSelector(PerpService.getSubaccountTotalUnrealizedPNLAndFundingFee.selector,
            ↪  aliceSubId),
            abi.encode(-int256(400 ether), 0)
    );

    (int256 totalUPNL2, ) = perpService.getSubaccountTotalUnrealizedPNLAndFundingFee(aliceSubId);

    int256 collateral2 = assetService.getSubaccountTotalMargin(aliceSubId) + totalUPNL2; // USDC

    // Assert that the margins are different
    console.log("collateral1", collateral1);
    console.log("collateral2", collateral2);
    assertTrue(collateral1 != collateral2);
}
```

Use this test in `assetService.setter.t.sol` to see the expected output.

Recommendation: Based on my current understanding aggregate realized and unrealized balances before applying collateral factors.

3.2 Medium Risk

3.2.1 Deposit transactions / events may mixed in case of chain re-org and lead to double deposit

Submitted by BengalCatBalu

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: According to `README`, the deposit process is as follows:

> When a user deposits assets (e.g., stablecoins), this event reflects the increase in their available balance. Backend servers confirm the transaction and emit this event on-chain. The smart contract updates the user's collateral state to ensure they can trade or maintain positions.

That is:

- User sends transactions to → `Vault` → `emit Deposit Event` → off-chain sequencer catches this event and send a transaction to `DepositHandler` → `DepositHandler` calls `AssetService.AdjustCollateral` → collateral increases.

But what if the following events were to take place and swap places: transaction from Sequencer to Deposit Handler, User sends transaction to Vault. In case of incorrect transaction control, off chain sequencer will allow user to make double deposit.

Finding Description: Let's take a look at how transaction control is performed. When Deposit is called in Vault:

```
emit LogDeposit(_requestId, _subaccount, _tokenAddress, _amount);
```

The `requestId` is responsible for identifying the transaction. Then, in the depositHandler, each requestId is processed as follows:

```
function _executeAction(bytes memory _data) internal override {
    (uint256 _requestId, bytes32 _subaccount, uint256 _amount, address _tokenAddress, DepositAction _action) =
        abi.decode(_data, (uint256, bytes32, uint256, address, DepositAction));
    if (_subaccount != bytes32(0) && !isProcessed[_requestId]) {
        if (_action == DepositAction.Process) {
            IAssetService(assetService).adjustCollateral(_subaccount, _tokenAddress, _amount.toInt256());
            emit LogDepositProcessed(_subaccount, _tokenAddress, _amount);
        } else {
            emit LogDepositRejected(_subaccount, _tokenAddress, _amount);
        }
        isProcessed[_requestId] = true;
    }
}
```

That is, each `requestId` is simply marked as fulfilled in the dictionary. Note that there is no binding to the `Vault` contract. Obviously, this control will not catch the case of re-org attack. Since Deposit Handler does not check whether such a requestId has been registered in Vault. Let's consider two scenarios, normal and c re-org.

An important condition for an incorrect count to occur is the presence of any other transaction that increments the request id in the same block as the sequencer transaction. But if we look at `Vault::deposit` process - there is no any permission for any user to deposit for any subaccount. So, this condition can also happen intentionally. But it is still more likely to occur by accident.

- Normal flow:
  - Block 1:
    - (1) `USER1` Deposit → generated `requestId = 1`.
  - Block 2:
    - (2) Off-chain Sequencer send deposit to `USER1` with `requestId = 1`.
    - (3) `DepositHandler` marks `requestId = 1` as completed, funds goes to `USER1`.
    - (4) `USER2` Deposit → generate `requestId = 1`.
- Re-org flow:
  - Block 1:
    - (1) Off-chain Sequencer send deposit to `USER1` with `requestId = 1`.
    - (2) `DepositHandler` marks `requestId = 1` as completed, funds goes to `USER1`.
    - (3) `USER2` Deposit → generate `requestId = 1` (then `DepositHandler` transaction for him will revert).
  - Block 2:
    - (4) `USER1` Deposit → generate `requestId = 2` (then he will receive his funds again, thus receiving double amount).

Impact Explanation: In case of chain re-org one user will receive his deposit * 2 while one other user will fail his deposit.

Likelihood Explanation: Likelihood is low, because off probability of chain re-org.

Recommendation: Add a protection mechanism for such cases, at least check on deposit handler, that `requestId` was generated.

3.2.2 Rejected deposit transaction to `DepositHandler` will cause the funds stuck in the Vault

Submitted by Aamirusmani1552, also found by KupiaSec, Roberto, BengalCatBalu, silverologist, aua-oo7, Bigsam, 0xEllipticCurve, CasinoCompiler, ZoA and zxriptor

Severity: Medium Risk

Context: Vault.sol#L99

Description: When User calls the `Vault::deposit(...)` function, his assets are transferred to the Vault and `requestId` is generated for the deposit that will be used to increase his collateral:

```
function deposit(address _tokenAddress, bytes32 _subaccount, uint256 _amount)
    external
    payable
    nonReentrant
    onlyAcceptedToken(_tokenAddress)
    returns (uint256 _requestId)
{
    // if depositing native token, then deposit to wrapped eth.
    if (msg.value > 0) {
        address _weth = WETH;
        _requestId = _deposit(_weth, _subaccount, msg.value);
        IWETH(_weth).deposit{ value: msg.value }(); // <<<
    } else {
        // transfer token to this contract
        _requestId = _deposit(_tokenAddress, _subaccount, _amount);
        IERC20(_tokenAddress).safeTransferFrom(msg.sender, address(this), _amount); // <<<
    }
}
```

But the sequencer can reject that transaction. The reason could be liquidatable account or some other reason. Because of this reject transaction, the `requestId` that was generated in the vault deposit is consumed:

```
function _executeAction(bytes memory _data) internal override {
    (uint256 _requestId, bytes32 _subaccount, uint256 _amount, address _tokenAddress, DepositAction _action) =
        abi.decode(_data, (uint256, bytes32, uint256, address, DepositAction));
    if (_subaccount != bytes32(0) && !isProcessed[_requestId]) {
        if (_action == DepositAction.Process) {
            IAssetService(assetService).adjustCollateral(_subaccount, _tokenAddress, _amount.toInt256());
            emit LogDepositProcessed(_subaccount, _tokenAddress, _amount);
        } else {
            emit LogDepositRejected(_subaccount, _tokenAddress, _amount); // <<<
        }
        isProcessed[_requestId] = true; // <<<
    }
}
```

And because of that, if now user try to withdraw his tokens using the `Vault::withdraw(...)` function, it will revert since the the withdraw function makes call to `WithdrawHandler::getWithdrawableAmount(...)` which will return zero as there was no collateral deposited in the user's position. Also there is no separate function to recover those tokens. They will remain stuck in the contract.

Proof of Concept:

```
// SPDX-License-Identifier: BUSL
pragma solidity 0.8.19;

// 3rd-party - interfaces
import { IERC20 } from "@openzeppelin/token/ERC20/IERC20.sol";
// 3rd-party - contract
import { Test } from "forge-std/Test.sol";
import { WETH } from "solmate/tokens/WETH.sol";
import { MockERC20 } from "solmate/test/utils/mocks/MockERC20.sol";
import { ProxyAdmin } from "@openzeppelin/proxy/transparent/ProxyAdmin.sol";
import { TransparentUpgradeableProxy } from "@openzeppelin/proxy/transparent/TransparentUpgradeableProxy.sol";
import { Vault } from "src/peripherals/Vault.sol";

// contracts
import { Entrypoint } from "src/Entrypoint.sol";
import { AssetService } from "src/services/AssetService.sol";
import { PerpService } from "src/services/PerpService.sol";
import { OracleService } from "src/services/OracleService.sol";
import { DepositHandler } from "src/handlers/DepositHandler.sol";
import { WithdrawHandler } from "src/handlers/WithdrawHandler.sol";
import { TradeHandler } from "src/handlers/TradeHandler.sol";
import { AdminHandler } from "src/handlers/AdminHandler.sol";
import { ExecutionFeeHandler } from "src/handlers/ExecutionFeeHandler.sol";
```

```
import { LiquidationHandler } from "src/handlers/LiquidationHandler.sol";
import { CollateralSeizeHandler } from "src/handlers/CollateralSeizeHandler.sol";
import { SwapHandler } from "src/handlers/SwapHandler.sol";

// interfaces
import { IPerpService } from "src/interfaces/services/IPerpService.sol";
import { IEntrypoint } from "src/interfaces/IEntrypoint.sol";
import { IHandler } from "src/interfaces/IHandler.sol";

// libs
import { DeployerUtil } from "test/utils/DeployerUtil.sol";
import "./OrderbookBase.t.sol";
import "src/handlers/DepositHandler.sol";

contract MyTests is OrderbookBaseTest {
    Vault internal vault;
    address internal sequencer = makeAddr("sequencer");

    function setUp() public override {
        super.setUp();

        // deploy vault
        address proxyAdmin = address(new ProxyAdmin());
        address _vault = address(new Vault());

        TransparentUpgradeableProxy proxy = new TransparentUpgradeableProxy(
            _vault, proxyAdmin, abi.encodeWithSelector(Vault.initialize.selector, address(withdrawHandler))
        );
        vault = Vault(payable(address(proxy)));

        vault.setMinDeposit(address(USDC), 1e6);
        vault.setMinDeposit(address(weth), 1e16);
        vault.setWETH(address(weth));
        vault.setWithdrawableToken(address(USDC), true);
        vault.setWithdrawableToken(address(weth), true);
        vault.setWhitelist(sequencer, true);
    }

    function test_rejectedDepositBalanceCannotBeWitdhrawnFromTheVault() public {
        uint256 amount = 1000 * 1e6;
        _dealAndApproveToHandler(alice, address(USDC), amount);

        // alice deposits to vault
        vm.startPrank(alice);
        USDC.approve(address(vault), amount);
        vault.deposit(address(USDC), aliceSubId, amount);
        vm.stopPrank();

        // sequencer rejects the deposit.
        // Reason could be liquidatable account, or any other reason.
        vm.prank(address(entrypoint));
        depositHandler.executeAction(
            abi.encode(1, aliceSubId, amount, address(USDC), DepositHandler.DepositAction.Reject)
        );

        // no collateral should be deposited into the assetService
        assertEq(assetService.collaterals(aliceSubId, address(USDC)), 0);

        // alice's balance should have been transferred to the vault
        assertEq(USDC.balanceOf(alice), 0);
        assertEq(USDC.balanceOf(address(vault)), amount);

        // alice tries to withdraw from the vault but the transaction reverts because of no collateral
        ↪  deposited
        // in the vault
        vm.startPrank(alice);
        vm.expectRevert(Vault.Vault_ExceedWithdrawableAmount.selector);
        vault.withdraw(address(USDC), aliceSubId, amount);
    }
}
```

Recommendation: It is recommended to add the emergence withdraw function, or give the user access to generate new requests to withdrawal separately.

3.2.3 Same Execution fee is used for different Settlement tokens

Submitted by Aamirusmani1552, also found by BengalCatBalu

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: As per the sponsors, any ERC20 could be used as a settlement token. But in the `ExecutionFeeHandler`, the max execution fee is set to `1e7` which is in `USDC` token. And there is no way to update this fee. That means, if some other token except `USDC` is used as a settlement token, the protocol will only be able to charge upto `1e7` of that token. Which would be just a dust amount for most of the tokens. Because of that, protocol might suffer a huge revenue loss.

Proof of Concept:

```
// SPDX-License-Identifier: BUSL
pragma solidity 0.8.19;

// 3rd-party - interfaces
import { IERC20 } from "@openzeppelin/token/ERC20/IERC20.sol";
// 3rd-party - contract
import { Test } from "forge-std/Test.sol";
import { WETH } from "solmate/tokens/WETH.sol";
import { MockERC20 } from "solmate/test/utils/mocks/MockERC20.sol";
import { ProxyAdmin } from "@openzeppelin/proxy/transparent/ProxyAdmin.sol";
import { TransparentUpgradeableProxy } from "@openzeppelin/proxy/transparent/TransparentUpgradeableProxy.sol";
import { Vault } from "src/peripherals/Vault.sol";

// contracts
import { Entrypoint } from "src/Entrypoint.sol";
import { AssetService } from "src/services/AssetService.sol";
import { PerpService } from "src/services/PerpService.sol";
import { OracleService } from "src/services/OracleService.sol";
import { DepositHandler } from "src/handlers/DepositHandler.sol";
import { WithdrawHandler } from "src/handlers/WithdrawHandler.sol";
import { TradeHandler } from "src/handlers/TradeHandler.sol";
import { AdminHandler } from "src/handlers/AdminHandler.sol";
import { ExecutionFeeHandler } from "src/handlers/ExecutionFeeHandler.sol";
import { LiquidationHandler } from "src/handlers/LiquidationHandler.sol";
import { CollateralSeizeHandler } from "src/handlers/CollateralSeizeHandler.sol";
import { SwapHandler } from "src/handlers/SwapHandler.sol";

// interfaces
import { IPerpService } from "src/interfaces/services/IPerpService.sol";
import { IEntrypoint } from "src/interfaces/IEntrypoint.sol";
import { IHandler } from "src/interfaces/IHandler.sol";

// libs
import { DeployerUtil } from "test/utils/DeployerUtil.sol";
import "./OrderbookBase.t.sol";
import "src/handlers/DepositHandler.sol";

contract MyTests is OrderbookBaseTest {
    Vault internal vault;
    address internal sequencer = makeAddr("sequencer");

    function setUp() public override {
        super.setUp();

        // deploy vault
        address proxyAdmin = address(new ProxyAdmin());
        address _vault = address(new Vault());

        TransparentUpgradeableProxy proxy = new TransparentUpgradeableProxy(
            _vault, proxyAdmin, abi.encodeWithSelector(Vault.initialize.selector, address(withdrawHandler))
        );
        vault = Vault(payable(address(proxy)));

        vault.setMinDeposit(address(USDC), 1e6);
        vault.setMinDeposit(address(weth), 1e16);
        vault.setWETH(address(weth));
        vault.setWithdrawableToken(address(USDC), true);
        vault.setWithdrawableToken(address(weth), true);
        vault.setWhitelist(sequencer, true);
    }
```

```
    function test_ExecutionFeeHandlerChargesFeeInUSDCForAnySettlementToken() public {
        uint256 fee = 0.01e18; // 0.01 WETH
        bytes memory data = abi.encode(aliceSubId, fee, protocolFeeSubaccount);

        assertEq(assetService.collaterals(aliceSubId, address(weth)), 0);
        assertEq(assetService.collaterals(protocolFeeSubaccount, address(weth)), 0);

        vm.prank(address(entrypoint));
        vm.expectRevert(ExecutionFeeHandler.ExecutionFeeHandler_FeeExceedsMax.selector);
        executionFeeHandler.executeAction(data);
    }
}
```

Recommendation: It is recommended to make the execution fee dynamic. Either make the execution fee variable updatable, or add a mapping for different tokens corresponding to their fees.

3.2.4 `WithdrawHandler.getWithdrawableAmount()` should utilize `_unsettled`, which is not scaled by the collateral factor

Submitted by KupiaSec, also found by trachev

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: When withdrawing `settlementToken`, the `getWithdrawableAmount()` function calculates the maximum withdrawable amount using the formula `collaterals + _unsettled`, where collaterals is the recorded state value in `AssetService`, and `_unsettled` includes values from `PnL` and `lending/borrowing` fees. The issue is that `_unsettled` is scaled by the collateral factor. The collateral factor should only be applied when checking the health factor, not during the withdrawal process. As a result, the withdrawable amount is miscalculated, preventing withdrawers from accessing the full amount they are entitled to, leading to locked funds.

Finding Description: The WithdrawHandler.getWithdrawableAmount() function accounts for `_unsettled`.

`_unsettled` is the sum of `PnL` and `lending/borrowing` fees, representing the amount the account should receive from or pay to the protocol. This value is calculated by the `_getSubaccountUnsettledE18()` function, as seen at line 96.

```
    function getWithdrawableAmount(bytes32 _subaccount, address _tokenAddress) public view returns (uint256
    ↪  _amount) {
        IAssetService _assetService = IAssetService(assetService);
        int256 _subaccountTotalBalance = _assetService.collaterals(_subaccount, _tokenAddress);
        address _settlementToken = _assetService.settlementToken();
        if (_tokenAddress == _settlementToken) {
            uint256 _decimals = _assetService.collateralTokenDecimals(_settlementToken);
96          int256 _unsettled = _getSubaccountUnsettledE18(_subaccount) / int256(10 ** (18 - _decimals));
            if (_unsettled > 0) {
98              _subaccountTotalBalance += _unsettled;
            }
        }
        _amount = _subaccountTotalBalance > 0 ? uint256(_subaccountTotalBalance) : uint256(0);
    }
```

However, in the `_getSubaccountUnsettledE18()` function, `_unsettledE18` is scaled by the collateral factor of the `settlementToken`, which is less than 1. The collateral factor is intended for checking the health factor. By applying the collateral factor, the calculated withdrawable amount is less than it should be, resulting in withdrawers being unable to withdraw the full amount they are entitled to.

```
    function _getSubaccountUnsettledE18(bytes32 _subaccount) internal view returns (int256 _unsettledE18) {
        IAssetService _assetService = IAssetService(assetService);
        address _perpService = perpService;
        (int256 _totalUPnLE18, int256 _totalFundingFeeE18) =
            IPerpService(_perpService).getSubaccountTotalUnrealizedPNLAndFundingFee(_subaccount);
        int256 _borrowingFeeE18 = _assetService.getSubaccountPendingBorrowingFee(_subaccount);

        _unsettledE18 = _totalUPnLE18 - _totalFundingFeeE18 - _borrowingFeeE18;

        // apply collateral factor if unsettled balance is positive
        if (_unsettledE18 > 0) {
153         _unsettledE18 = _unsettledE18 *
↪   _assetService.collateralFactors(_assetService.settlementToken()).toInt256()
                / BPS.toInt256();
        }
    }
```

Impact Explanation: High. Using the collateral factor results in a lower withdrawable amount, preventing withdrawers from accessing the full amount they are entitled to. Consequently, users' collateral cannot be fully withdrawn, and a portion will remain locked in the protocol.

Likelihood Explanation: High. This issue occurs whenever withdrawals are processed.

Recommendation: The `getWithdrawableAmount()` function should not use `_getSubaccountUnsettledE18()` to calculate the unsettled amount. Instead, it should utilize the unsettled amount that is not scaled by the collateral factor.

3.2.5 Borrowing fee calculation is inconsistent

Submitted by Amar Fares

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `AssetService::_settlePendingBorrowingFee()` function is called when the user's settlement balance is updated or it's needed for a calculation.

```
function _settlePendingBorrowingFee(bytes32 _subaccount) internal {
    IAssetService.BorrowingAndLendingInfo memory _currentInfo = getCurrentProtocolBorrowingAndLendingInfo();
    IAssetService.BorrowingAndLendingInfo memory _userInfo = borrowingAndLendingInfos[_subaccount];
    // if _subaccount's rates info is up-to-current, early return.
    if (
        _currentInfo.borrowingRateE18 == _userInfo.borrowingRateE18
            && _currentInfo.lendingRateE18 == _userInfo.lendingRateE18
    ) {
        return;
    }

    int256 _settlementTokenBalance = collaterals[_subaccount][settlementToken];

    if (_settlementTokenBalance > 0) {
        _payLendingFeeToSubaccount(
            _subaccount, _userInfo.lendingRateE18, _currentInfo.lendingRateE18, _settlementTokenBalance
        );
    } else if (_settlementTokenBalance < 0) {
        _receiveBorrowingFeeFromSubaccount(
            _subaccount, _userInfo.borrowingRateE18, _currentInfo.borrowingRateE18, _settlementTokenBalance
        );
    }
    // update _subaccount's rates info to current rates info
    borrowingAndLendingInfos[_subaccount] = _currentInfo;
}
```

Let's look at the scenario when a user's settlement token balance is in the negative, where `_receiveBorrowingFeeFromSubaccount()` is called:

```
function _receiveBorrowingFeeFromSubaccount(
    bytes32 _subaccount,
    uint128 _userBorrowingRateE18,
    uint128 _protocolBorrowingRateE18,
    int256 _negativeSettlementAssetBalance
) internal {
    int256 _borrowingFee = uint256(_protocolBorrowingRateE18 - _userBorrowingRateE18).toInt256()
        * _negativeSettlementAssetBalance / 1e18;
    _adjustCollateral(_subaccount, settlementToken, _borrowingFee);
    emit LogSubaccountPayBorrowingFee(_subaccount, uint256(-_borrowingFee));
}
```

The borrowing fee to be paid is the difference between the current protocol borrowing rate and the last borrowing rate checkpoint the user has, multiplied by their balance. The protocol's borrowing rate is pushed manually by the protocol using `AdminHandler.sol`. But due to the way it is calculated, a user might be charged more/less depending on when this function is called. Let's look at an example:

- Alice:
  - Alice's balance is -100.
  - Rate jumps from 10% → 20%, function is called and has to pay 10 (Balance at -110 now).
  - Rate jumps from 20% → 30%, function is called and she has to pay 11 (Balance at -121 now).
  - Paid: 21, Balance: -121.
- Bob:
  - Bob's balance is -100.
  - Rate jumps from 10% → 20% but Bob does not interact with the protocol so function is not called, other people can't call it for Bob either.
  - Rate jumps from 20% → 30% Bob now interacts with the protocol, he still pays 20% but that is -20 in this case.
  - Paid: 20, Balance: -120.

Both users were taxed the same rate and for the same amount, but if the user did not interact with the protocol during the period between the first and second rate hike, they will pay a smaller fee. This will just lead to the protocol calculations getting messed up and in the long-term the protocol will suffer losses from it.

Recommendation: Accumulate the fees in a different manner so they add up, regardless if the user interacted with the protocol during that time or not.