BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

SmarDex

USDN Ecosystem

January 2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	SmarDex — USDN Ecosystem
Website	Smardex.io
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/SmarDex-Ecosystem/usdn-contracts/tree/5ac07c31764beb7948290b97569f4e1750bcf6e6
Resolution 1	https://github.com/SmarDex-Ecosystem/usdn-contracts/tree/aa7a166c963944ec6f43a2533be1577886f73830
Resolution 2	https://github.com/SmarDex-Ecosystem/usdn-contracts/commit/f26db82370dbcbeddf3f36fb561424d31b3f8351

2. Detection Overview

Low and Informational issues have been removed since the team was only interested in High and Medium issues for this round.

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	6	4		2
Medium	50	6		44
Low	-			
Informational	-			
Governance	2			2
Total	58	10		48

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Libraries

DoubleEndedQueue

The DoubleEndedQueue library is heavily inspired by OpenZeppelin's DoubleEndedQueue library:

(https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/DoubleEndedQueue.sol)

It allows for adding, removing and clearing initiated actions from the queue. Contrary to the contract logic which uses both ends of the queue, new actions are pushed to the end and executed actions are removed from the queue via three different scenarios:

a) The initiate action is at the beginning of the queue: Increment the queue begin and pop the first element from the queue.

b) The initiate action is at the end of the queue: Decrement the queue end and pop the last element from the queue.

c) The initiate action is in the middle of the queue: Zero out the element and remove it from the queue once the queue begin increases.

The core logic which alters the queue is handled within the ProtocolCore contract, therefore, we have created an appendix in that section.

This library is exclusively used by the ProtocolCore contract.

No issues found.

TickMath

The TickMath library is responsible for calculating the tick at a price and vice-versa. It is used throughout the architecture to determine the liquidation price from a position. Similar to UniswapV3, it leverages the logarithm to achieve the conversion goal.

No issues found.

LiquidationRewardsManager

LiquidationRewardsManager

The LiquidationRewardsManager contract is an externally deployed contract which inherits the ChainlinkOracle and Ownable2Step contracts. The main purpose of this contract is the calculation and return of the wstETH amount corresponding to a liquidation action.

This is done by incorporating the configured reward parameters and the additional actions taken, such as rebases and rebalances.

Whenever a liquidation is happening, the following actions are (optionally) executed:

a) Ticks are liquidated
b) Protocol is rebalanced
c) USDN is rebased

based on which action(s) are executed and the provided gas price, the refund is calculated.

The refund calculation is based on the overall amount of gas used, which is derived as follows:

otherGasUsed + BASE_GAS_COST + (gasUsedPerTick * liquidatedTicks)

Additionally, a rebase and rebalancer execution will increase the gas used amount as follows:

+ rebaseGasUsed

+ rebalancerGasUsed

The used gas amount is then multiplied with 2 wei and the specific gasMultiplier which is 10_500 BPS. A fixed reward in ETH which is 0.001 ETH comes on top of that.

The contract does furthermore incorporate a bonus which is depending on the price deviation between the currentPrice and the tickPrice of the liquidated tick(s) as well as the totalExpo of a tick which has been liquidated.

Issue_01	Wrong wstETH conversion within getLiquidationRewards
Severity	**Medium**
Description	The getLiquidationRewards function is flawed on multiple occasions: a) reward in ETH is increased by stETH: totalRewardETH += _wstEth.getStETHByWstETH(wstEthBonus); b) reward in ETH is converted incorrectly to wstETH: wstETHRewards_ = _wstEth.getWstETHByStETH(totalRewardETH); If ETH != stETH, this will yield an incorrect reward amount.
Recommendations	Consider: a) Converting the wstEthBonus to ETH b) Converting the totalRewardETH first to stETH and then to wstETH Alternatively, this issue can be acknowledged if it is assumed that stETH corresponds to the ETH value (which is however not granted).
Comments / Resolution	Acknowledged.

Issue_02	Maximum reward can be bypassed via single liquidations
Severity	**Medium**
Description	The maximum reward for a liquidation process is 0.5 ETH and depending on various factors, most importantly it depends on the price difference of the current price and the liquidation price of a position. Therefore, it is theoretically possible to receive the 0.5 ETH reward upon the liquidation of a single tick. Since usually most of the time, more than 1 tick is liquidated, that will result in aggregating the liquidation reward. However, this can be bypassed by liquidating only one tick at a time.
Recommendations	Consider forcing users to liquidate all liquidatable ticks at once. While that may expose gas-related issues, one can implement a governance functionality which allows for liquidating single ticks in such a scenario. Moreover, this fix will also fix other issues in the codebase.
Comments / Resolution	Acknowledged.

OracleMiddleware

ChainlinkOracle

The ChainlinkOracle contract is a simple oracle handler contract which is inherited by the OracleMiddleware contract. It is responsible for safely fetching the ETH price.

For the inheritance with the OracleMiddleware contract, the following functions will be used:

- _getFormattedChainlinkLatestPrice
- _getFormattedChainlinkPrice

These functions will return the value scaled to 18 decimals (middlewareDecimals) including the confidence interval, which is then used throughout the protocol.

No issues found.

PythOracle

The PythOracle contract is a simple oracle handler contract which is inherited by the OracleMiddleware contract and is responsible for providing low-latency data. Compared to Chainlink, the Pyth Oracle is a push-based oracle which means that users need to update the price on their own, rather than just fetching the price.

The main idea behind the usage of the Pyth Oracle is the possibility to fetch fast and continuous price-movements.

There are two scenarios where the Pyth price is fetched:

a) For the low latency scenario
b) As fallback oracle for the initiation scenario in case the latest Pyth update is fresher than the latest Chainlink update

Similarly to the Chainlink Oracle, the price is returned, scaled to 18 decimals.

In the scenario where an action is **initiated**, the parsePriceFeedUpdates function is used which will fetch the latest update in the past 45 seconds.
In the scenario where an action is **validated** (within 20 minutes), it will fetch the earliest update since the initiation time and initiation time + 20 minutes via the parsePriceFeedUpdatesUnique function. More context can be found in the official Pyth documentation: https://docs.pyth.network/price-feeds/api-reference/evm/parse-price-feed-updates-unique

No issues found.

OracleMiddleware

The OracleMiddleware contract is an auxiliary contract which is considered by the main protocol whenever the ETH price is needed. It is extended by the WstEthOracleMiddleware contract.

The main functionality of this contract is the parseAndValidatePrice function which is considered on the following interactions:

- liquidation (liquidate)
- initialization (initialize)
- opening initiation (_prepareInitiateOpenPositionData)
- closure initiation (_prepareClosePositionData)
- opening validation (_prepareValidateOpenPositionData)
- closure validation (_validateClosePositionWithAction)
- deposit initiation (_prepareInitiateDepositData)
- deposit validation (_validateDepositWithAction)
- withdraw initiation (_prepareWithdrawalData)
- withdraw validation (_validateWithdrawalWithAction)

As mentioned, this contract evolves mainly around the parseAndValidatePrice function, which fetches prices from Chainlink and Pyth, depending on the current circumstances and situation. It inherits all two oracle handler contracts (Chainlink and Pyth).

For initiation actions, the logic is as follows:

a) If a data parameter is provided, fetch most recent Pyth price

b) If no data parameter is provided, fetch latest stored Pyth price and latest stored Chainlink price and return the most recent one

For validation actions, the logic is as follows:

a) If the execution is within initiateTimestamp + 20 min, fetch the Pyth price adjusted for the CI
b) If the targetLimit (initiateTimestamp + 20 min) is surpassed, the Chainlinkprice after that time is considered

Furthermore, governance can change various parameters.

Issue_03	Governance Privilege: Variable settings
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For instance, the _validationDelay can be changed to any value which will result in fetching a price at a different timestamp (for validations)
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_04	_validateChainlinkRoundId will revert in case of phaseId increase between roundId and latestRoundId
Severity	**High**
Description	The _validateChainlinkRoundId function validates the return data of the chainlink oracle. A special validation mechanism was incorporated in the scenario where the roundId corresponds to a new phaseId. More information about this mechanism can be found here: https://bailsec.io/tpost/8oi4p03381-about-chainlink-oracle-roundid-phaseid-a This case is handled as follows: *(,,, uint256 previousRoundTimestamp,) = _priceFeed.getRoundData(roundId - 1);* *// if the provided round's timestamp is 0, it's possible the aggregator recently changed and there is no data* *// available for the previous round ID in the aggregator. In that case, we accept the given round ID as the* *// sole reference with additional checks to make sure it is not too far from the target timestamp* *if (previousRoundTimestamp == 0) {* *// calculate the provided round's phase ID* *uint80 roundPhaseId = roundId >> 64;* *// calculate the first valid round ID for this phase* *uint80 firstRoundId = (roundPhaseId << 64) + 1;* *// the provided round ID must be the first round ID of the phase, if not, revert* *if (firstRoundId != roundId) {* *revert OracleMiddlewareInvalidRoundId();* *}* *(uint80 latestRoundId,,,,) = _priceFeed.latestRoundData();*

```
        // if the provided round ID does not belong to the latest phase,
revert
        if (latestRoundId >> 64 != roundPhaseId) {
            revert OracleMiddlewareInvalidRoundId();
        }

        // make sure that the provided round ID is not newer than it
should be
        if (providedRoundPrice_.timestamp > targetLimit +
_timeElapsedLimit) {
            revert OracleMiddlewareInvalidRoundId();
        }
```

As mentioned in the comment, if previousRoundTimestamp is zero, that means that roundId just experienced an aggregatorId update which means roundId-1 points to invalid data. In that scenario, the whole above mentioned validation mechanism is executed.

At the end of the validation mechanism, the following check is executed:

```
        (uint80 latestRoundId,,,,) = _priceFeed.latestRoundData();
        // if the provided round ID does not belong to the latest phase,
revert
        if (latestRoundId >> 64 != roundPhaseId) {
            revert OracleMiddlewareInvalidRoundId();
        }
```

This is problematic in the scenario where another phaseId update has happened between [roundId; latestRoundId], as this check will now always revert, resulting in the whole price fetching mechanism to revert and thus validations to revert.

The inherent assumption that the current phaseId has to match with the provided roundId's phaseId is flawed.

Recommendations	Consider removing this check. It furthermore must be ensured that the removal of this check doesn't open any other potential leaks.
Comments / Resolution	Resolved.

Issue_05	Insufficient validation within setLowLatencyDelay can result in faulty state
Severity	**Medium**
Description	Currently, there is no sufficient validation for the setLowLatencyDelay function. A general invariant within this architecture is that lowLatencyDelay > lowLatencyDeadline. If that invariant is violated, it is possible for users to prevent executions to take out the securityDeposit (see issue within the first audit re: risk free trades). This invariant is nicely ensured within the following spot: *if (newLowLatencyValidatorDeadline > lowLatencyDelay) {* *revert IUsdnProtocolErrors.UsdnProtocolInvalidValidatorDeadline();* *}* (see Setters library) However, within the OracleMiddleware, such a check is non-existing: *function setLowLatencyDelay(uint16 newLowLatencyDelay) external onlyRole(ADMIN_ROLE) {* *if (newLowLatencyDelay < 15 minutes) {* *revert OracleMiddlewareInvalidLowLatencyDelay();*

	`` } if (newLowLatencyDelay > 90 minutes) { revert OracleMiddlewareInvalidLowLatencyDelay(); } _lowLatencyDelay = newLowLatencyDelay; emit LowLatencyDelayUpdated(newLowLatencyDelay); } ``` For example it can now happen that the deadline can be set to 19 minutes and delay to 15 minutes, which then violates the invariant.
Recommendations	Consider implementing such a check within the setLowLatencyDelay function.
Comments / Resolution	Resolved.

WstEthOracleMiddleware

The WstEthOracleMiddleware contract is an extension of the OracleMiddleware contract which converts the received ETH price into wstEth by using the stEthPerToken function within the wstETH contract.

Issue_06	StETH depeg will result in incorrect price
Severity	**Medium**
Description	The parseAndValidatePrice function converts the price from the ETH oracle to wstETH by using _wstEth.stEthPerToken(). This function returns the stETH amount for 1e18 wstETH. In the scenario where 1 ETH != 1 stETH, the returned price will be incorrect.
Recommendations	Consider using the stETH oracle feed instead of the ETH oracle feed.
Comments / Resolution	Acknowledged. The Smardex team came to the conclusion that the ETH oracle is more reliable and will thus keep the current practice.

Rebalancer

Rebalancer

The Rebalancer contract is a fundamental instrument for the USDN protocol. It is an auxiliary contract which allows users to deposit funds which are then aggregated into a single leveraged long position.

The core protocol relies on imbalance checks which ensure that neither the vault nor the long side is majorly dominant at any point in time. This is ensured by various imbalance checks for all entry positions. For example, if the vault side is already in dominance, deposits may be forbidden.

Since liquidations inherently force-close positions which results in a decrease of the long side and an increase of the vault side, it can happen that liquidations bring the protocol into an imbalanced state. It is furthermore clear that there cannot be any imbalance checks which prevent liquidations because liquidations are mandatory for the correct functionality of the protocol.

Therefore, the Rebalancer contract was developed which is triggered whenever liquidations have happened which could bring the protocol into an imbalanced state with the intention to open a leveraged long position which encounters the imbalance towards the vault side.

Users that deposit into the rebalancer versus the core protocol will have the following benefits:

a) No fee is applied for opening a long position
b) A rebalancer bonus is distributed everytime positions are liquidated and the rebalancer is triggered. This bonus goes towards the rebalancer's long position

Similarly to the core protocol, the rebalancer incorporates an initiate/validate system.

The initiateDepositAssets function allows users to initiate their deposit assets which can then be validated via the validateDepositAssets function. Once validated, these flow into the _pendingAssetsAmount variable which marks them ready for the next rebalancer interaction.

Appendix: EntryAccMultiplier

Whenever a new position is created, all current pending assets from users will be allocated to this position and the current multiplier is stored to this position. The initial multiplier is 1e38 and will change based on the position value upon the opening and the next update. To illustrate this, consider the following example:

a) Position is opened:
-> amount = 10e18
-> multiplier = 1e38

b) Position is closed and reopened (has gained PnL/funding value in the meantime):
-> position amount after closure = 20e18
-> accumulator: 20e18 * 1e38 / 10e18
-> 2e38

The mathematical formula for the accumulator calculation is as follows:

> previousPosValue * previousPosData.entryAccMultiplier / previousPosData.amount

Whereas previousPosValue is the position value after the closure, entryAccMultiplier was the previous accumulator and previousPosData.amount was the position value at the opening.

If now users withdraw from their position, the to received amount is calculated as follows:

> amount * entryAccMultiplierNow / entryAccMultiplierThen

whereas entryAccMultiplierNow is the multiplier from the most recent position version and entryAccMultiplierThen is the multiplier from the entryPositionVersion (the first version where a deposit is being processed towards the USDN protocol).

Therefore, the balance adjustment is reflected by the adjusted accumulator.

Appendix: Bonus Application

Whenever positions are liquidated, the remaining collateral for all positions, which is ideally 2% of the total provided collateral at the beginning, will be transferred to the vault side, while a specific percentage part which is defined by rebalancerBonusBps, is being added to the position value of the rebalancer and thus removed from the vault side.

Whenever the rebalancer is updated, it is updated with the following values:

> _pendingAssetsAmount + previousPosValue

Within these amounts, the rebalancer bonus is not included.

The actual long position collateral in the USDN protocol is however computed as follows:

> _pendingAssetsAmount + previousPosValue + bonus

As we can see, a discrepancy between the value in the rebalancer and the value in the USDN protocol is existent, this discrepancy properly reflects the bonus value. This value can never become negative, thus underflows in the calculation wont be possible.

If users now attempt to withdraw, the bonus will be applied on their attempted withdrawal amount, pro-rata on the attempted withdrawal amount to the overall balance in the rebalancer:

> amountToCloseWithoutBonus + amountToCloseWithoutBonus * (amountLongInUSDN - amountInRebalancer) / amountInRebalancer

In such a scenario where the bonus is not claimed or only partially claimed, it will flow into the accumulator gain for the next round, offsetting any eventual losses or increasing gains.

The contract furthermore incorporates logic which allows for delegating away closures to other addresses via a signed message.

Issue_07	Edge-case within initiateDeposit assets allows malicious user to erroneously erase validated and non-processed deposit
Severity	**High**
Description	Within the Rebalancer, the initiateDepositAssets function allows users to initiate a new deposit which can then either be validated or reset later: *function initiateDepositAssets(uint88 amount, address to) external nonReentrant {* */* authorized previous states:* *- not in rebalancer* *- amount = 0* *- initiateTimestamp = 0* *- entryPositionVersion = 0* *- included in a liquidated position* *- amount > 0* *- 0 < entryPositionVersion <= _lastLiquidatedVersion* *OR* *- positionData.tickVersion != protocol.getTickVersion(positionData.tick)* **/* *if (to == address(0)) {* *revert RebalancerInvalidAddressTo();* *}* *if (amount < _minAssetDeposit) {* *revert RebalancerInsufficientAmount();* *}* *UserDeposit memory depositData = _userDeposit[to];* *// if the user entered the rebalancer before and was not liquidated* *if (depositData.entryPositionVersion > _lastLiquidatedVersion) {* *uint128 positionVersion = _positionVersion;*

```
        PositionData storage positionData =
_positionData[positionVersion];
        // if the current position was not liquidated, revert
        if (_usdnProtocol.getTickVersion(positionData.tick) ==
positionData.tickVersion) {
                revert RebalancerDepositUnauthorized();
        }

        // update the last liquidated version and delete the user data
        _lastLiquidatedVersion = positionVersion;
        delete depositData;
        } else if (depositData.entryPositionVersion > 0) {
        // if the user was in a position that got liquidated, we should reset
the deposit data
        delete depositData;
        } else if (depositData.initiateTimestamp > 0 ||
depositData.amount > 0) {
        // user is already in the rebalancer
        revert RebalancerDepositUnauthorized();
        }

        depositData.amount = amount;
        depositData.initiateTimestamp = uint40(block.timestamp);
        _userDeposit[to] = depositData;

        _asset.safeTransferFrom(msg.sender, address(this), amount);

        emit InitiatedAssetsDeposit(msg.sender, to, amount,
block.timestamp);
        }
```

Once validated, it will then be used for the next rebalancer update.

A problem is existing with the following scenario:

```
        // if the user entered the rebalancer before and was not
liquidated
        if (depositData.entryPositionVersion > _lastLiquidatedVersion) {
        uint128 positionVersion = _positionVersion;
        PositionData storage positionData =
_positionData[positionVersion];
        // if the current position was not liquidated, revert
        if (_usdnProtocol.getTickVersion(positionData.tick) ==
positionData.tickVersion) {
                revert RebalancerDepositUnauthorized();
        }

        // update the last liquidated version and delete the user data
        _lastLiquidatedVersion = positionVersion;
        delete depositData;
```

This scenario should handle cases where the user has already an existing rebalancer position and this position is considered as liquidated.

However, an unconsidered edge-case is exposed here which is based on the fact that a user can have an already validated position which was not yet processed, which means the position is due for the next update. In the scenario where the current version of the rebalancer was liquidated:

```
if (_usdnProtocol.getTickVersion(positionData.tick) ==
positionData.tickVersion) {
        revert RebalancerDepositUnauthorized();
}
```

It is now possible to deposit on behalf of the user who has the unprocessed, validated position and erroneously overwrite the deposit data:

	delete depositData; *...* *depositData.amount = amount;* *depositData.initiateTimestamp = uint40(block.timestamp);* *_userDeposit[to] = depositData;* This will essentially result in a loss of funds for the previous existing, unprocessed position.
Recommendations	Consider simply reverting if there is an unprocessed deposit, e.g. if depositData.entryPositionVersion > _positionVersion
Comments / Resolution	Resolved. An important observation must be noticed that the fix is not 100% correct. In the scenario where a user has a processed position which is below _positionVersion and now the currentTick is liquidated, a user cannot invoke initiateDepositAssets until the rebalancer is updated. The Smardex team added the following comment: The implemented fix introduces a minor side effect. While addressing this side effect was considered, we could not fully ensure that doing so would not inadvertently introduce additional vulnerabilities. Given that the side effect does not pose any risk to users' funds, we have made the informed decision to acknowledge this issue rather than implement further changes.

Issue_08	Rebalancer can remain locked due to frequent rebalances
Severity	**Medium**
Description	The Rebalancer contract exposes a _closedLockedUntil variable which is set to block.timestamp + 4 hours whenever a rebalance has happened. Users in the rebalancer can then only withdraw their processed position once that period has surpassed. In times of frequent rebalances, that means users may never be able to withdraw due to this lock.
Recommendations	Consider carefully observing the protocol and eventually adjust the lock time downwards in such scenarios.
Comments / Resolution	Acknowledged.

Issue_09	Possible lost funds due to accMultiplier of zero
Severity	**Medium**
Description	A fundamental mechanism is the calculation of how much funds users will receive once they withdraw their deposited amount. This is handled via the accMultiplier. Whenever the first rebalance happens or whenever the previous position was liquidated, the next position update will set entryAccMultiplier to 1e38. Once the next rebalance then happens, the entryAccMultiplier will be adjusted based on the initial position value and the current position value. We have elaborated this mechanism in-depth in the corresponding appendix. A rare side effect can occur if the multiplier is in a downwards spirale which means with every update it decreases (without liquidations) until it eventually reaches zero. If now new positions are validated and subsequently processed while at the same time the entryAccMultiplier remains zero, it will become impossible for these depositors to withdraw their share because of a division by zero: *data.amountToCloseWithoutBonus = FixedPointMathLib.fullMulDiv(* *data.amount,* *data.currentPositionData.entryAccMultiplier,* ***_positionData[data.userDepositData.entryPositionVersion].entryAccMultiplier*** *);*
Recommendations	Consider carefully handling the scenario where the multiplier becomes zero, ensuring no other side-effects are introduced.

Comments / Resolution	Failed resolution, a new high risk issue has been introduced. We recommend reverting the implementation to its previous version acknowledging this issue in an effort to prevent the introduction of new side effects. Furthermore, the practice of liquidating the existing position as soon as the multiplier falls < 10_000 will have the side-effect that large positions which have just been deposited are completely liquidated. **Resolution 2:** Acknowledged. The _triggerRebalancer function has been adjusted which acknowledges this issue.

UsdnProtocol/libraries

ActionsLongLibrary

The ActionsLong library implements functionality for opening and closing long positions. A special mechanism of position initiation and position validation was implemented with the rationale to counter any potential price-frontrunning attacks. Whenever a position is initiated, the initiation time is saved for the validation, during the validation, it will then attempt to use the initiation timestamp + 24 seconds to get the oracle price.

Notably, the following functionality is exposed:

Opening Initiation: This is the first function which is invoked upon the position creation. This function initiates the position opening which allows users to provide wstETH and create a leveraged long position. While this opened position is already considered a fully functional long position which manipulates and experiences funding, it still needs to be validated with the final validation price. As usual, this action is pushed into the _pendingActionsQueue, waiting to be picked up by the validator.

Opening Validation: Whenever a position opening has been initiated, it is sitting within the _pendingActionsQueue, waiting to be validated by the validator or other users at shortest 24 seconds after the initiation timestamp. The validation will uses the new price for the totalExpo calculation while ensuring that the funding which has been experienced between initiation and validation is still applied. Moreover, any PnL is being reversed.

In the case where the maxLeverage is exceeded due to a price decrease or funding application, a special scenario is triggered which is explained in “Appendix: MaxLeverage Excess”.

Closure Initiation: Whenever a position opening has been fully validated, the position can then be again fully or partially closed. The closure initiation already fully removes a position from the system or decreases it, which includes updating all corresponding global data plus the tick where the position was stored.

This means the position will not experience any further funding nor will it influence the funding calculation.

The PendingAction will be pushed to the queue which can then be picked up by the validator or any other user, users will only receive their corresponding position value once the closure has been validated.

Closure Validation: Whenever a position closure has been initiated and is in the queue, it can be picked up 24 seconds after the initiation by the validator or any other address. Upon the position closure, the position value is recalculated based on the updated price and the applied fee. If the current price is above or equal the liquidation price, the position value will not be paid out to the recipient but it will allocated towards the vault side, if not liquidated, the position value will be paid out to the recipient and any potential deviation is applied/deducted to the vault side (see Appendix: Position Closure Deviation). Since the position is already fully closed and does not impact funding anymore, no funding will be applied.

The pending action is then removed from the queue.

Appendix: Position Value Determination

Whenever a new position is added, the position value is stored in Position.amount and reflects the provided wstETH amount from a user upon position opening. This will be considered as collateral and flows into the _longBalance state variable. Any leverage will be considered as "borrowed amount".

This borrowed amount is not stored into an own state variable but is rather the product of the subtraction from _totalExpo - _balanceLong, for all positions.

If desired to calculate it for only one position, the formula is: position.totalExpo - position.amount.

It is of utmost importance that the accumulator and PnL invariants both hold. Adjustments/Unadjustments must never be changed during position modifications or PnL applications.

A position value is calculated as follows:

a) Calculate the adjusted liquidation price of a position, this will either be larger, smaller or equal to the provided initial liquidation price, based on the experienced funding rate since the position creation.

   > liqPriceNoPenaltyUnadjusted * (currentPrice * tradingExpo) / accumulator

   The logic behind this calculation is clear, if _balanceLong increases, this will decrease tradingExpo and therefore the new liquidationPrice for a position will be lower. The position will thus be liquidated later and worth more.

   In reverse, if _balanceLong decreases, this will decrease tradingExpo, making the liquidationPrice higher and therefore the position will be liquidated earlier and will be worth less.

b) Calculate the position value:
   > position.totalExpo * (currentPrice - liquidationPriceWithoutPenaltyAdjusted) / currentPrice

A real world example can be found under “Appendix: PnL calculation & Liquidation impact”

Appendix: LiqMultiplierAccumulator

Whenever a position is added, removed or liquidated, the _liqMultiplierAccumulator state variable is changed. This variable represents the **sum of totalExpo * unadjustedTickPriceNoPenalty** for all positions.
Since both totalExpo and unadjustedTickPrice are variables which are never affected by funding payments or changes in price (PnL), the _liqMultiplierAccumulator does not have to be updated for funding or PnL adjustments.

The accumulator is used in “adjusting” and “unadjusting” positions.
The **unadjusted price** is the price corresponding to the **tick a position is stored at**, while the **adjusted price** is the **liquidation price of a position/real price of a position**, updated by funding since position creation.

This means to determine the liquidation price, the price must be unadjusted

The formula for unadjusting a position is:

unadjustedPrice = adjustedPrice * accumulator / (assetPrice * (totalExpo - balanceLong))

Vice-versa, the formula for adjusting a position is:

adjustedPrice = unadjustedPrice * (assetPrice * (totalExpo - balanceLong)) / accumulator

Illustrated this means the following:

a) Upon position opening, users provide their desired liquidation price. This price is then converted to the unadjustedPrice and **forms the tick where the position is stored at.**

b) Upon liquidations, the currentPrice is fetched and converted to the unadjustedPrice. **This forms the tick, above which, liquidations are happening.**

c) Upon position value calculation, the stored tick is converted to the corresponding price and adjusted.

Now we will elaborate the math behind this logic:

a) Whenever a new position is created, the accumulator will be increased as follows:

   liquidationPrice * pos.totalExpo

b) Whenever the unadjustedPrice is calculated, this will be done as follows:

   currentPrice * accumulator / [currentPrice * tradingExpo]

Following this math, we can quickly realize the following: The accumulator increase is from the same value as the divisor side increase, as (if the protocol has matured, the same ratio will be kept by using unadjustedTickNoPenalty):

liquidationPrice * pos.totalExpo = currentPrice * tradingExpo

This therefore means, whenever positions are created, liquidated or closed, the accumulator increase or decrease should always reflect the correct change of [currentPrice * tradingExpo].

To further meet the goal of keeping the ratio, instead of the liquidationPrice, the unadjustedTick (without penalty) is used. If for example funding was +- zero, that means that the ratio/precision remains +- 1.0000

Consider the following example:

Position creation without fee

- price = 3000e18
- amount = 1e18
- desiredLiqPrice = 1530e18

a) Calculate pos.totalExpo:
   -> amount * price / (price-liquidationPriceWithoutPenalty)
   -> 1e18 * 3000e18 / (3000e18-1500e18)
   -> 2e18
b) Calculate accumulator increase:
   -> liquidationPriceWithoutPenalty * pos.totalExpo
   -> 1500e18 * 2e18
   -> 3e39
c) Apply our result in the unadjustedPrice calculation:
   -> price * accumulator / (currentPrice * tradingExpo)
   -> 3000e18 * 3e39 / (3000e18 * 1e18)
   -> 3000e18
d) This invariant is therefore mathematically proven

Appendix: Opening validation

Whenever a position opening has been initiated, this position already impacts funding and has the impact on the protocol as if the position is already fully validated.

However, in fact this position must first be validated using the price at initiationTimestamp + 24 seconds. Since the position already impacts the funding rate, it must also experience the said.

To facilitate this, the _validateOpenPosition function calculates the position exposure using the old liquidationMultiplier which then in turn results in an overall position value that is impacted by the funding. It is important to note that the nominal funding which is experienced on the initiated position will not be 1:1 reflected on the validated position. Instead, the validated position will experience the funding as if it would be active since the initiation timestamp. This means if the exposure for the validated position is higher, the funding will be higher, the same counts for the vice-versa case.

Moreover, any PnL which has been experienced will be reversed via _validatePositionUpdateBalances function. If for example the price significantly increased and the initiated position contributes to an increase of balanceLong, this increase will be reversed.

Appendix: MaxLeverage Excess

Whenever a position is initiated, it is ensured that the maximum leverage cannot be larger than the desired parameter (usually 10x). This is checked using the price at the initiation timestamp.

Since the position validation usually uses the price at initiationTS + 24s, there is a small risk that the price significantly drops with 24s which would increase the leverage. The Smardex team has implemented a special solution for that scenario, where the new liquidation price is calculated based on the validation price and the maximum leverage.

The old position is then fully closed and a new position is opened with the target that the funding is still applied while any PnL is reversed.

Appendix: Position Closure Deviation

As we have already explained, a position closure consists of initiation and validation and importantly, the price upon the validation is considered to determine how much a user will actually receive upon closure. Therefore, there are two scenarios:

a) Users will receive less tokens due to a price decrease: Since the position is already fully closed and out of the system, the difference will be simply allocated to the vault side.

b) Users will receive more tokens due to a price increase: Since the position is already fully closed and out of the system, the difference will be simply taken from the vault side.

A third scenario is existent in the case where the price falls below the liquidation price of the position, in that case the whole position value during the initiation time will be allocated to the vault.

The sophisticated logic behind these calculations is that the vault increase/decrease is accurately equal to the PnL change which would have been applied using the “old” tradingExpo (before the position closure initiation).

Issue_10	Flaw within maxLeverage excess scenario results in incorrect funding application
Severity	**High**
Description	Within the _validateOpenPositionWithAction function, it is possible that the leverage of a position exceeds the maxLeverage due to a price decrease between the initiation and validation time. In such a scenario where the maxLeverage is exceeded, the new liquidation price is calculated based on the maxLeverage and the current price. The goal of this methodology is to simply create a new position while applying the funding on the position and reversing any potential PnL. A problem however lies within the fact how the new expo and position value is calculated: *data.pos.totalExpo =* *Utils._calcPositionTotalExpo(data.pos.amount, data.startPrice, data.liqPriceWithoutPenalty);* *// adjust the balances to reflect the new value of the position* *uint256 updatedPosValue =* *Utils._positionValueOptimized(data.pos.totalExpo, data.lastPrice, data.liqPriceWithoutPenalty);* As one can see, updatedPosValue is calculated based on liqPriceWithoutPenalty, which has now the effect that the funding is actually not applied and instead the vault will bear the funding. **PoC written:** Status Quo:

> balanceVault = 1e18
> balanceLong = 0
> price = 3000e18
> fee = 0%
> liqMultiplierAcc = 1e38
> penalty = 198
> tickSpacing = 1

a) Alice initiates position opening
> desiredLiqPrice = 2754e18
> adjustedPrice = 3000e18
> unadjustedTickWithPenalty = 79212
> adjustedLiqPriceWithoutPenalty = 2700e18
> amount * startPrice / (startPrice - liqPriceNoPenaltyAdjusted)
> 1e18 * 3000e18 / (3000e18 - 2700e18)
> positionTotalExpo = 10e18
> accumulator + (liqPriceNoPenaltyUnadjusted * positionExpo)
> (2700e18 * 10e18)
> liqMultiplierAccumulator = 2.7e40
> storage:
> balanceVault = 1e18
> balanceLong = 1e18
> totalExpo = 10e18
> liqMultiplierAccumulator = 2.7e40
> pending pos:
> tick = 79212

b) price decreases from 3000e18 to 2999e18 and pnl is applied + protocol experiences positive funding

> funding = 0.1e18
> totalBalance = 2e18
> balanceLong + (tradingExpo * priceDiff / newPrice) - fundingFee
> 1e18 + (9e18 * (-1e18) / 2999e18) - 0.1e18
> balanceLongWithPnLAndFunding = 0.896e18

> totalBalance - tempLongBalance
> 2e18 - 0.896e18
> balanceVaultWithPnLAndFunding = 1.104e18
> totalExpo - balanceLong
> 10e18 - 0.896e18
> tradingExpo = 9.104e18

c) position is validated
> startPrice = 2999e18
> not liquidated; no liqPending

> liqPriceNoPenaltyUnadjusted * (currentPrice * tradingExpo) / accumulator
> 2700e18 * (2999e18 * 9.104e18) / 2.7e40
> liqPriceWithoutPenaltyAdjusted = 2730.289e18

> position.totalExpo * (currentPrice - liquidationPriceNoPenaltyAdjusted) / currentPrice
> 10e18 * (2999e18 - 2730.289e18) / 2999e18
> oldPosValue = 0.896e18
> liqPriceWithoutPenaltyNorFunding = 2700e18

> decimals * startPrice / (startPrice - liqPrice)
> 1e18 * 2999e18 / (2999e18 - 2700e18)
> leverage = 10.003e18 ; exceeds maxLeverage
> maxLeverage < leverage scenario
> startPrice - ((DECIMALS * startPrice) / leverage)
> 2999e18 - ((1e18*2999e18) / 10e18)
> liqPriceWithoutPenalty = 2699,1e18
 > price * decimals / liqMultiplier
> tempTickWithoutPenalty = 79010
 > tickWithPenalty = 79208
 > liqPriceWithoutPenalty = 2699.1e18
> now remove old position
 > totalExpo = 0

> accumulator - (liqPriceNoPenaltyUnadjusted * totalExpoToRemove)
> liqMultiplier = 0

> amount * startPrice / (startPrice - liqPriceNoPenaltyAdjusted)
> 1e18 * 2999e18 / (2999e18 - 2699.1e18)
> pos.totalExpo = 10e18
> save new position
> totalExpo = 10e18
> accumulator + (liqPriceNoPenaltyUnadjusted * positionExpo)
> (2699.1e18 * 10e18)
> liqMultiplier = 2.6991e40

> position.totalExpo * (currentPrice - liquidationPriceNoPenaltyAdjusted) / currentPrice
> 10e18 * (2999e18 - 2699.1e18) / 2999e18
> updatedPosValue = 1e18
> _validateOpenPositionUpdateBalances
> diff = newPosValue - oldPosValue
> 1e18 - 0.896e18
> diff = 0.104e18
> balanceVault - diff
> 1.104e18 - 0.104e18
> 1e18
> balanceLong + diff
> 0.896e18 + 0.104e18
> 1e18
> Due to erroneous change, lets now calculate the value of the position:
> get adjustedPrice
> liqPriceNoPenaltyUnadjusted * (currentPrice * tradingExpo) / accumulator
> 2699.1e18 * (2999e18 * 9e18) / 2.6991e40
> 2699.1e18
> get pos value
> position.totalExpo * (currentPrice -

liquidationPriceNoPenaltyAdjusted) / currentPrice
> 10e18 * (2999e18 - 2699.1e18) / 2999e18
> 1e18

PoC coded:

```
function testNewPositionFundingValueIncorrectCalculation() public {
        vm.prank(ADMIN);
        protocol.setFundingSF(10 **
Constants.FUNDING_SF_DECIMALS);
        // leverage approx 10x
        (, PositionId memory posId) = protocol.initiateOpenPosition(
        uint128(LONG_AMOUNT),
        CURRENT_PRICE * 9 / 10,
        type(uint128).max,
        protocol.getMaxLeverage(),
        address(this),
        payable(this),
        type(uint256).max,
        abi.encode(CURRENT_PRICE),
        EMPTY_PREVIOUS_DATA
        );
        (PendingAction memory pendingAction,) =
protocol.i_getPendingAction(address(this));
        LongPendingAction memory longPendingAction =
protocol.i_toLongPendingAction(pendingAction);
        uint128 firstTickPrice =
protocol.getEffectivePriceForTick(posId.tick);

        uint128 validationPrice = CURRENT_PRICE - 100 ether;
        uint128 newLiqPrice =
protocol.i_getLiquidationPrice(validationPrice,
uint128(protocol.getMaxLeverage()));
        (int24 validationTick,) = protocol.i_getTickFromDesiredLiqPrice(
        newLiqPrice, longPendingAction.liqMultiplier,
```

```
protocol.getTickSpacing(), protocol.getLiquidationPenalty()
        );
        assertLt(validationTick, posId.tick, "tick");
        PositionId memory newPos = PositionId(validationTick, 0, 0);
        uint128 newTickPriceBefore =
protocol.getEffectivePriceForTick(validationTick);

        int newVaultBalanceWithJustInitialPositionFundingAndPNL;
        {
        (Position memory position,) =
protocol.getLongPosition(initialPosition);
        console.log("initialPositionExposure",position.totalExpo);

        int initialPositionLongBalance = 5e18;
        int initialPositionExposure = 9919970269703463156;

        int priceDiff = int(uint256(validationPrice)) -
int(uint(CURRENT_PRICE));

        int intialPositionPNL = int(initialPositionExposure -
initialPositionLongBalance) * priceDiff / int(uint256(validationPrice));

        int fund = 101287932229482;
        int initialPositionFunding = fund * int(initialPositionExposure -
initialPositionLongBalance) / 10**18;

        uint256 vaultbalance = protocol.getBalanceVault();
        newVaultBalanceWithJustInitialPositionFundingAndPNL =
int(vaultbalance) + initialPositionFunding - intialPositionPNL;
        }

        _waitDelay();

        vm.expectEmit();
        emit LiquidationPriceUpdated(posId, newPos);
```

	(, PositionId memory newPosId) = protocol.validateOpenPosition(payable(this), abi.encode(validationPrice), EMPTY_PREVIOUS_DATA); *// check that all ticks have now a higher price* *assertGt(protocol.getEffectivePriceForTick(validationTick), newTickPriceBefore, "new tick price");* *assertGt(protocol.getEffectivePriceForTick(posId.tick), firstTickPrice, "first tick price");* *// new vault balance doesn't have the funding of the validated position. it instead only has funding + pnl of the other position* *// if instead, the actualCurrentPrice was used for valuing the updated position, then the vault balance will also have the funding of the validated position consistent with the non-maxLeverage scenario* *{* *uint256 vaultbalance = protocol.getBalanceVault();* *assert(vaultbalance == uint(newVaultBalanceWithJustInitialPositionFundingAndPNL));* *}* *}*
Recommendations	Since the maxLeverage excess scenario was already flawed during the first audit iteration and now remains flawed after the 2nd audit, our only recommendation is to simplify the code and remove it. We understand that this will result in a potential excess leverage in case the price drops significantly after 24 seconds. However, after some visualizations, we are of the opinion that this is an acceptable downside.
Comments / Resolution	Acknowledged. The Smardex team added the following comment: Although the maximum leverage scenario presents certain limitations,

	we consider it to be an essential safety measure. Eliminating this scenario would have allowed positions to assume risks beyond acceptable thresholds, potentially leading to outcomes that are both more hazardous and unpredictable compared to the side effect described. Consequently, we have opted to acknowledge the issue in favor of maintaining overall system stability.

Issue_11	Potential underflow due to erroneous position value calculation
Severity	**Medium**
Description	We have already elaborated above a flaw within the maxLeverage excess scenario. The same root-cause of the incorrect updatedPosValue calculation can result in an underflow: *uint256 updatedPosValue = Utils._positionValueOptimized(data.pos.totalExpo, data.lastPrice, data.liqPriceWithoutPenalty);* On the first view, it doesn't seem possible that data.liqPriceWithoutPenalty > data.lastPrice, since a lot of sanity checks are already execute within the preparation, including the following: *uint128 liqPriceWithPenalty = Utils._getEffectivePriceForTick(data_.action.tick);* *// A user that triggers this condition will be stuck in a validation loop until it liquidates its own position* *// with the stored `_lastPrice`* *if (data_.lastPrice <= liqPriceWithPenalty) {* *data_.isLiquidationPending = true;* *return (data_, false);* *}* A problem however occurs that here liqPriceWithPenalty includes the

	actual funding which has occurred between [initiation; validation], which means that liqPriceWithPenalty can actually largely deviate from liqPriceWithoutPenalty and be lower (in case of negative funding). If that happens, it is now theoretically possible that the liquidation check passes but the actual calculation for the updatedPosValue: *position.totalExpo * (lastPrice - liqPriceNoPenaltyNoFunding) / lastPrice* underflows, if liqPriceNoPenaltyNoFunding > lastPrice
Recommendations	Since the maxLeverage excess scenario was already flawed during the first audit iteration and now remains flawed after the 2nd audit, our only recommendation is to simplify the code and remove it. We understand that this will result in a potential excess leverage in case the price drops significantly after 24 seconds. However, after some visualizations, we are of the opinion that this is an acceptable downside.
Comments / Resolution	Resolved.

Issue_12	Closure initiation with lastPrice does not align with validation
Severity	**Medium**
Description	The closure initiation uses the lastPrice for the slippage check. The _lastPrice is however always the neutral price and this does not align with the currentPrice.price usage within the closure validation: data.priceWithFees = (currentPrice.price - currentPrice.price * s._positionFeeBps / Constants.BPS_DIVISOR).toUint128(); This means the actual slippage check via userMinPrice is not completely accurate.
Recommendations	Consider if that is acceptable, if not consider switching to the currentPrice including the confidence interval.
Comments / Resolution	Acknowledged.

Issue_13	Funding rate inconsistency during position closures
Severity	**Medium**
Description	Whenever a position is initiated for closure, the funding fee is extrapolated with the intention to effect a user's position up to block.timestamp: *data_.longTradingExpo = Core.longTradingExpoWithFunding(data_.lastPrice, uint128(block.timestamp));* A problem in this scenario is that the actual core update may not be up to block.timestamp but rather a timestamp in the past. At the same time, it must be acknowledged that the actual position was opened up to block.timestamp which means it should indeed impact the funding up to this time. If the position is now closed without updating the core state up to block.timestamp, it will happen that the funding update between [_lastUpdateTimestamp; block.timestamp] does not include the removed position.
Recommendations	Consider force-updating the state with the most recent block.timestamp.
Comments / Resolution	Acknowledged.

Issue_14	Vanilla liquidation within closure validation can imbalance the protocol without rebalancing
Severity	**Medium**
Description	The position closure validation ignores any funding but applies the price at initTS + 24s to calculate the accurate position value to be removed: *data.positionValue = Utils._positionValue(* *long.closePosTotalExpo,* *data.priceWithFees,* *Utils._getEffectivePriceForTick(tickWithoutPenalty, long.liqMultiplier)* *);* Based on the new position value, the vault will either receive assets from the removed position or pay towards the user. All of these interactions will imbalance the protocol without any imbalance check. The same counts for the scenario where the price drops below the position's liquidation price: *if (currentPrice.neutralPrice <= data.liquidationPrice) {* *// position should be liquidated, we don't transfer assets to the user* *// position was already removed from tick so no additional bookkeeping is necessary* *// credit the full amount to the vault to preserve the total balance invariant* *s._balanceVault += long.closeBoundedPositionValue;* *emit IUsdnProtocolEvents.LiquidatedPosition(* *long.validator, // not necessarily the position owner* *Types.PositionId({ tick: long.tick, tickVersion:*

	long.tickVersion, index: long.index }), *currentPrice.neutralPrice,* *data.liquidationPrice* *);* *return (false, true);* *}*
Recommendations	Consider rebalancing the protocol in case this practice results in imbalance.
Comments / Resolution	Acknowledged.

Issue_15	Opening validation liquidation check is not including extrapolated funding
Severity	**Medium**
Description	Whenever a position is validated for opening, the following liquidation check is executed: *if (data_.lastPrice <= liqPriceWithPenalty) {* *data_.isLiquidationPending = true;* *return (data_, false);* *}* liqPriceWithPenalty corresponds to the adjusted price based on the current contract state: *uint128 liqPriceWithPenalty =* *Utils._getEffectivePriceForTick(data_.action.tick);* Moreover, the contract state is updated based on the oracle return value.

	If the return value is below block.timestamp, the contract does not account for all funding which has happened since block.timestamp and the adjusted liqPriceWithPenalty does neither. This can result in a user skipping liquidations.
Recommendations	Consider extrapolating the funding up to block.timestamp for the liquidation price check.
Comments / Resolution	Acknowledged.

Issue_16	PnL reset can be exploited by vault depositors to gain a benefit
Severity	**Medium**
Description	Within the opening validation, part of the design is that the PnL is being reset and the funding fee is still applied. Any PnL reset will alter balanceVault which can then in turn be abused by users to deposit/withdraw. Moreover, the closure validation can also result in balanceVault change which can be abused the same way.
Recommendations	There is no trivial fix with the current design implementation.
Comments / Resolution	Acknowledged.

Issue_17	Lack of opening fee caching can change outcome
Severity	**Medium**
Description	Whenever a position is initiated for opening, the slippage check is being executed as follows: *uint256 adjustedPrice =* *(data_.lastPrice - data_.lastPrice * s._positionFeeBps / Constants.BPS_DIVISOR).toUint128();* *if (adjustedPrice < params.userMinPrice) {* *revert IUsdnProtocolErrors.UsdnProtocolSlippageMinPriceExceeded();* *}* Once the position is then validated, the new price is calculated as follows: *data_.startPrice =* *(currentPrice.price + currentPrice.price * s._positionFeeBps / Constants.BPS_DIVISOR).toUint128();* If the fee has been changed between [initiation; validation] the user will pay a different fee than expected which will manipulate the outcome.
Recommendations	Consider caching the fee in a similar fashion as within the vault.
Comments / Resolution	Acknowledged.

Issue_18	Penalty change for maxLeverage scenario can result in exceeding maxLeverage
Severity	**Medium**
Description	Within the maxLeverage scenario, the possibility exists that the new tick where the position is meant to be stored has a different penalty than the protocol penalty in the scenario where there are already existing positions and the protocol penalty has changed in the meantime. This will result in a different liqPriceWithoutPenalty than expected which can then either result in excess leverage or decreased leverage.
Recommendations	Consider keeping that in mind
Comments / Resolution	Acknowledged.

ActionsUtilsLibrary

The ActionsUtils library is an utility library which exposes functionality for many different purposes, it is leveraged throughout the long opening and closure process. Explicit functionalities are:

- Liquidate positions
- Transferral of position ownership
- Helper logic for closure initiation
- Execution of actionable actions

Appendix: Validation periods

The validateActionablePendingActions function allows users to validate an initiated action in the following timeframes and claim the security deposit:

a) Between 15 and 20 minutes post initiation
b) After 85 minutes post initiation

Issue_19	Incorrect clamping during position closure initiation due to lack of extrapolation application on balanceLong
Severity	**Medium**
Description	Within the position closure flow, longTradingExpo is extrapolated as follows (ActionsUtilsLibrary): data_.longTradingExpo = Core.longTradingExpoWithFunding(data_.lastPrice, uint128(block.timestamp)); At the same time, the positionValue is clamped if it exceeds balanceLong: } else if (uint256(positionValue) > balanceLong) This exposes an inconsistency because on one hand, the state is extrapolated but on the other hand, balanceLong is not adjusted for extrapolation which can result in a user receiving less tokens than he should (based on extrapolation assumption)
Recommendations	We do not recommend a change since the change on _balanceLong would actually apply the extrapolation. However, this issue should be carefully documented.
Comments / Resolution	Acknowledged.

Issue_20	Lack of fee application during position closure initiation will result in inaccurate imbalance check
Severity	**Medium**
Description	The _prepareClosePositionData is responsible for preparing important data towards the position closure, including the calculation of the position value which will be removed. The position value is calculated using the current price: *// the approximate value position to remove is calculated with `_lastPrice`, so not taking into account* *// any fees. This way, the removal of the position doesn't affect the liquidation multiplier calculations* *// to have maximum precision, we do not pre-compute the liquidation multiplier with a fixed* *// precision just now, we will store it in the pending action later, to be used in the validate action* *int24 tick = Long._calcTickWithoutPenalty(s, posId.tick, data_.liquidationPenalty);* *data_.tempPositionValue = _assetToRemove(* *s,* *data_.lastPrice,* *Long.getEffectivePriceForTick(tick, data_.lastPrice, data_.longTradingExpo, data_.liqMulAcc),* *data_.totalExpoToClose* *);* As one can see within the highlighted comment, this is desired at this stage. A problem however occurs due to the fact that this position value is then used to calculate any potential vault imbalance which can happen due to the position closure. In reality however, while the

position value will still be the same “to be removed value”, the fee will not be transferred to the recipient but rather allocated to the vault side:

```
if (assetToTransfer < long.closeBoundedPositionValue) {
uint256 remainingCollateral;
unchecked {
remainingCollateral = long.closeBoundedPositionValue -
assetToTransfer;
}

s._balanceVault += remainingCollateral;
```

It has to be noted that the fee is taken during the validation step and allocated towards the vault side.

Consider the following incorrect scenario

balanceLong = 50e18
totalExpo = 100e18
balanceVault = 50e18
tradingExpo = 50e18

> close position with 5e18 fully, no fee
> (50e18 - 45e18) * 10000 / 45e18
> 1111

Consider the following correct scenario

balanceLong = 50e18
totalExpo = 100e18
balanceVault = 50e18
tradingExpo = 50e18

	> close position with 5e18 fully, 10% fee goes to vault > (50.5e18 - 45e18) * 10000 / 45e18 > 1222
Recommendations	Consider incorporating the fee into the imbalance check.
Comments / Resolution	Acknowledged.

Issue_21	Slippage check uses price without CI
Severity	**Medium**
Description	In a similar fashion as the previously raised issue where the slippage check is done against _lastPrice without the fee, the slippage check is now done against _lastPrice (including the fee) but without the confidence interval. This exposes an inconsistency against the validation flow.
Recommendations	Consider documenting this issue.
Comments / Resolution	Acknowledged.

Issue_22	Incorrect callback parameter within ownershipCallback
Severity	**Medium**
Description	Within the transferPositionOwnership function, a callback is executed to the newOwner address with msg.sender as param. Since callback parameter is msg.sender instead of the old owner and msg.sender will be the router, the parameter will be incorrect.
Recommendations	Consider using the old owner address instead of msg.sender (router).
Comments / Resolution	Resolved.

ConstantsLibrary

The ConstantsLibrary contains all important constant variables which are used throughout the protocol.

No issues found.

CoreLibrary

The CoreLibrary contract is used as a helper contract due to the modular approach of the codebase. It exposes logic for the following parts:

a) Contract initialization: After the deployment, governance initializes the contract and creates a deposit and long position. This is handled within the ProtocolCore's initialize function.

b) Funding calculation: The EMA and corresponding funding calculation is completely handled within this core contract, this functionality takes up most of the contract. *Due to multiple flaws, this will be completely refactored.*

c) PnL calculation: Similar to the funding calculation, the core contract handles the PnL calculation and applies profits/losses if the price changes.

d) Pending Action Logic: The core contract exposes the necessary logic to add, fetch and remove actions from the queue. Additionally it exposes a function that allows governance to remove pending actions which are blocking the queue.

Issue_23	Opening and SDEX fee will be lost if pending action is removed
Severity	**Medium**
Description	Whenever users deposit into the vault or open a long position, a fee is taken from the user (in the vault deposit, an additional SDEX fee on top). In the scenario where these actions are considered as blocked and removed, any fee will essentially remain lost.
Recommendations	Consider documenting this scenario.
Comments / Resolution	Acknowledged.

Issue_24	Funding rate update frequency impacts outcome
Severity	**Medium**
Description	The current implementation of the _fundingPerDay function adds the current imbalance to the existing EMA value in each update: *if (oldVaultExpo > oldLongExpo_) {* *denominator = uint256(oldVaultExpo * oldVaultExpo);* *fundingPerDay_ = -int256(* *FixedPointMathLib.fullMulDiv(* *numeratorSquared,* *s._fundingSF * 10 ** (Constants.FUNDING_RATE_DECIMALS - Constants.FUNDING_SF_DECIMALS),* *denominator* *)* *) + ema;* *} else {*

	*denominator = uint256(oldLongExpo_ * oldLongExpo_);* *fundingPerDay_ = int256(* *FixedPointMathLib.fullMulDiv(* *numeratorSquared,* *s._fundingSF * 10 ** (Constants.FUNDING_RATE_DECIMALS - Constants.FUNDING_SF_DECIMALS),* *denominator* *)* *) + ema;* calling the update function more frequently can cumulatively yield a higher total funding charge over the same time period. This effectively makes the total paid funding dependent on the frequency of updates rather than purely on how long the system was imbalanced and how large that imbalance was.
Recommendations	Consider documenting this.
Comments / Resolution	Acknowledged.

Issue_25	Removal of blocked closure validation does not account for new price
Severity	**Medium**
Description	Whenever a closure validation position is removed via _removeBlockedPendingAction, the position value is transferred out to the “to” address: *else if (pending.action == Types.ProtocolAction.ValidateClosePosition && cleanup) {* *// for pending closes, the position is already out of the protocol* *Types.LongPendingAction memory close = Utils._toLongPendingAction(pending);* *// send the value of the position at the time of the initiate to the `to` address* *address(s._asset).safeTransfer(to, close.closeBoundedPositionValue);* *// as the assets were already removed from the long's balance, there are no additional steps needed* *}* This behavior is an inconsistency compared to the usual closure validation as the initial position value is used and not the position value at initTS + 24s
Recommendations	Consider documenting this scenario.
Comments / Resolution	Acknowledged.

Issue_26	Invariant violation due to maxLongBalance clamping
Severity	**Medium**
Description	In such a scenario where balanceLong cannot be further increased due to the clamping: *uint256 maxLongBalance = _calcMaxLongBalance(s._totalExpo);* *if (data_.tempLongBalance > 0 &&* *uint256(data_.tempLongBalance) > maxLongBalance) {* *data_.tempLongBalance = maxLongBalance.toInt256();* *}* this will violate important invariants in the adjustment/unadjustment calculation. An important invariant of the protocol is that the liquidation price of a position should not be changed by any PnL. This can be illustrated as follows: pos.Expo = 10e18 balanceLong = 1e18 currentPrice = 3000e18 liqPriceUnadjustedWithoutPenalty = 2700e18 accumulator = 2.7e40 a) Price increases to 3300e18 balanceLong = 1.818e18 liqPriceNoPenaltyUnadjusted * (currentPrice * tradingExpo) / accumulator 2700e18 * (3300e18 * (10e18-1.818e18)) / 2.7e40

	> 2700e18 **As one can see, a PnL has not altered the liquidation price of the position which is due to the fact that the multiplier (currentPrice * tradingExpo) stays consistent as currentPrice increases and at the same time, tradingExpo decreases.** If now however, the PnL application does not alter balanceLong and thus tradingExpo remains its consistency while the price increases, the price increase will *push* the liquidation price up. We need to acknowledge that this price is only used to calculate the position value and due to the math, it has no impact on the said: *position.totalExpo * (currentPrice - liquidationPriceNoPenaltyAdjusted) / currentPrice* However, the fact that the liquidation price of a position is increased in that scenario, should be kept in mind.
Recommendations	Consider keeping this invariant violation and eventual side-effects in mind.
Comments / Resolution	Acknowledged.

Issue_27	Removing actions without cleanup can result in several side effects
Severity	**Medium**
Description	The removeBlockedPendingActionNoCleanup reverts initiated states. However, if that is done without cleanup it can have severe side-effects such as forever increased vault balances which will erroneously impact funding and imbalances.
Recommendations	Consider only using this functionality if there is no other solution.
Comments / Resolution	Acknowledged.

Issue_28	Stuck securityDeposit in case of blocked removal without cleanup
Severity	**Medium**
Description	Within the _removeBlockedPendingAction, the securityDeposit is only transferred if cleanup = true: *if (cleanup) {* *// slither-disable-next-line arbitrary-send-eth* *(bool success,) = to.call{ value: pending.securityDepositValue }("");* *if (!success) {* *revert IUsdnProtocolErrors.UsdnProtocolEtherRefundFailed();* *}* *}* In any other scenario, the securityDeposit is never transferred out which means it remains permanently stuck.

Recommendations	Consider only using the function with cleanup = false in emergency scenarios and be aware of this potential issue.
Comments / Resolution	Acknowledged.

Issue_29	Removal of blocked actions can imbalance the protocol
Severity	**Medium**
Description	The removeBlockedPendingActions function reverses previously initiated positions. Since the reverse of any position will inadvertently impact the protocol imbalance without any imbalance check, it is possible that post-removal the protocol is within an imbalanced state, without any subsequent rebalancing. For example in the scenario where an initiated deposit is removed, this will force-imbalance the protocol towards the long side.
Recommendations	Consider rebalance the protocol in such a scenario.
Comments / Resolution	Acknowledged.

Issue_30	Inconsistency in removing initiated opening
Severity	**Medium**
Description	Whenever an initiated opening is removed, contrary to the normal opening validation, the PnL is not reset but the user will actually gain any PnL which has been experienced since the position initiation. This includes the fact that the initiated price is used as base for any PnL calculation. Furthermore, the removal does not incur any fee nor confidence interval.
Recommendations	Consider if this is an acceptable case, if not consider reverting the PnL and only account for funding.
Comments / Resolution	Acknowledged.

LongLibrary

The ProtocolLong library handles most logic around long positions, such as fetching long positions from the storage, handling price adjustments and unadjustments, the liquidation process and the rebalancer activation, as well as the full core update.

Below we will enumerate each important process in-depth.

Furthermore, there are several helper functions which are invoked by other contracts with regards to position opening and closing as well as imbalance checks.

Appendix: Liquidation Process

The liquidation process is a fundamental instrument within the USDN architecture and forcefully closes long positions if the current price touches or goes below a corresponding tick. Since positions are aggregately stored at their corresponding ticks, a liquidation is not executed on each single position but on each liquidatable tick, which will then automatically liquidate each position in this tick.

A liquidation is triggered upon all initiations and validations as well as directly via the liquidate function. The protocol does not allow to initiate or validate a position if there is any outstanding liquidation.

This is how a liquidation unfolds:

First of all, the closest tick at the unadjustedPrice is determined, this tick serves as a threshold for when ticks are considered as liquidatable, in simple terms: All ticks which are equal or above this tick are liquidatable, all ticks which are below this tick are not liquidatable.

After the threshold has been determined, the _highestPopulated tick and its corresponding index within the Bitmap is fetched. Under normal circumstances, this will be the last set index within the Bitmap, however, if that is not the case, the next set index below is being fetched.

A loop is being executed as long as:

a) Provided iteration assignment is not reached

b) The next set index is above or equal the threshold

Within this loop, all cached parameters are adjusted accordingly to the tick liquidation and the included positions, such as:

- totalExpoToRemove: The amount being removed from the _totalExpo variable, contains the aggregated expo of all positions.

- accumulatorValueToRemove: The amount being removed from the _liqMultiplierAccumulator, in an effort to keep the adjustment and unadjustment result the same as before.

- _liquidatedPositions: The amount of positions which have been residing in the liquidated tick, this is then subtracted from _totalLongPositions to keep an adequate reflection of how many long positions are currently open.

- remainingCollateral: The leftover value from all positions within the liquidated tick, this is then removed from _balanceLong and added to _balanceVault to properly reflect the closure of all positions.

Afterwards, the contract state is updated based on these cached parameters.

Appendix: Price Adjustments

Within the USDN math, there are two distinct price conversions: Price Unadjustment and Price Adjustment. Below we will explain the rationale behind both, where each is used, how each is impacted and the formulas:

Price Unadjustment: The price unadjustment logic is used for two purposes:

- Determining the price/tick where a position is stored at
- Determining the global liquidation value

Since the protocol math is developed in such a way that all position values are aggregated together, a solution was crafted to be able to still determine distinct positions and their related

gains/losses based on the time they were created and the protocol state change since the creation.
Therefore, whenever a position is created, the desired liquidation price will be unadjusted and the position is stored at this unadjusted price. In simple words the rationale behind this is to ensure that the position value will be treated fairly, as any protocol impact can now be mathematically applied on this position.

The formula for unadjustments whenever **a position is created** is as follows:

`desiredLiqPrice * accumulator / (currentPrice * tradingExpo)`

The formula for unadjustments on the **global liquidation price** is as follows:

`currentPrice * accumulator / (currentPrice * tradingExpo)`

That shows us, whenever `tradingExpo` experiences an increase, the liquidation price will decrease, thus positions will be liquidated earlier. In reverse, whenever `tradingExpo` decreases, liquidations will happen later.

Due to the fact that each position will be stored at its "current" unadjusted price, historical state changes will not impact this position, but future impacts will do so.

Price Adjustment: The price adjustment logic is used for two purposes:

- Determine the value of a single position
- Determine the value of all positions in a single tick

If we circle back to the position creation logic, we remember that the nominal provided liquidation price was unadjusted using the following calculation:

`desiredLiqPrice * accumulator / (currentPrice * tradingExpo)`

If now the price is immediately adjusted back, the following calculation will be used:

`liqPriceNoPenaltyUnadjusted * (currentPrice * tradingExpo) / accumulator`

Using both formulas, it is clear, if the defined accumulator invariant is ensured and no expo change has happened, the unadjusted price will be converted back to the initial provided liquidation price.

If PnL is applied, this will not have any impact on the adjustment/unadjustment but the price change will ensure that the position size is changed properly.

If funding is positive, this means the tradingExpo increases, thus the product increases and the real liquidation price increases, which then decreases the position value. Vice versa, the liquidation price decreases and the position value increases

Appendix: TickSpacing

Positions are always stored at their corresponding liquidation price but instead at the nominal price, they are stored at ticks. The concept of ticks originates from UniswapV3 and their concentrated liquidity math, whereas users can add liquidity between ticks. When it comes to the topic of ticks, the variable tickSpacing is important to understand. This variable determines which ticks are valid. For Smardex, each tick is spaced 0.01% apart from the other tick. This means that each 0.01% price change is corresponding to a new tick. Similar as in Uniswap, this will become problematic in terms of gas.
In UniswapV3 this means that users could add liquidity to each consecutive tick which then results in swaps being unimaginably expensive because each tick needs to be considered for a larger swap. Smardex is faced with the same issue when it comes to liquidations because that would mean a liquidation needs to consider every single tick which would then eventually run out of gas and prevent successful liquidations.

To solve this issue, the tickSpacing concept was introduced which determines the ticks which are allowed to be “populated”. In Smardex, tickSpacing will be 100 which translates to roughly a change of 1.005% (1.0001^(100) - 1). Therefore, only each 100th tick will be populated, which means in the current setup of

MIN_TICK = -322_378

MAX_TICK = 980_000

The first populatable tick will be -322_300 and from there onwards each 100th tick can be populated. This means positions are stored with roughly 1.005% difference instead of 0.01%

difference, which will significantly decrease the gas cost for liquidations. The downside of such a large tickSpacing setting is limited.

Appendix: Fixed Precision Multiplier

To understand this Appendix, we highly recommend reading the LiqMultiplierAccumulator Appendix.

Whenever a position is being closed, a "snapshot" is taken from the current contract state and stored into the LongPendingActions struct which is then used to store the pending action. The position value calculation is always done using the adjusted price with the following formula:

liqPriceNoPenaltyUnadjusted * (currentPrice * tradingExpo) / accumulator

position.totalExpo * (currentPrice - liquidationPriceNoPenaltyAdjusted) / currentPrice

While the position value is already calculated within the closure initiation, it will also be calculated during the closure validation using the fresh price. Since the contract state during the initial closure is used to determine the position value, any funding application or position changes since the initiation must not be incorporated into the position value calculation, which means that the ratio of

currentPrice * tradingExpo / accumulator

must be the same as during the init state.

Hence the contract will simply store this ratio into the LongPendingAction struct as closeLiqMultiplier, being calculated as follows:

currentPrice * tradingExpo * 1e38 / accumulator

This will then be applied once the position closure is validated to ensure that the contract state in terms of funding will still be the same as during the closure initialization.

Appendix: Liquidation Price calculation from tradingExpo

The _calcLiqPriceFromTradingExpo is responsible for calculating the liquidation price from the provided tradingExpo. This is done to facilitate the correctness of the tick calculation where the rebalancer position is stored based on the desired targetTradingExpo.

To derive the formula for calculating liqPriceNoPenaltyAdjusted, we need to rearrange the given equation:

totalExpo = amount * startPrice / (startPrice - liqPriceNoPenaltyAdjusted)

to the following equation via derivation:

liqPriceNoPenaltyAdjusted = startPrice * (totalExpo - amount) / totalExpo

Appendix: Rebalancer

Whenever positions are liquidated, this will force close these positions and thus decrease the overall tradingExpo. During regular position closure, an imbalance check is executed which ensures that the protocol still stays in an equilibrium after the position is closed. This check is as follows:

(vaultBalance - tradingExpo) * 10000 / tradingExpo

As one can see, it is ensured that the tradingExpo drop does not bring the protocol out of equilibrium.

With liquidations however, there is no such imbalance check, naturally due to the fact that liquidations must happen under all circumstances. Therefore, it will happen that liquidations bring the protocol into an imbalanced state due to the fact that the tradingExpo will inherently decrease. The threshold for the imbalance state lies at 6%. This means if liquidations result in a 6% dominance of the vault side, then the rebalancer is triggered. The goal of the rebalancer is to minimize the vault side dominance.

Issue_31	Incorrect longBalance reset in case of outstanding bad-debt position
Severity	**High**
Description	The _handleNegativeBalances function sets the longBalance to zero in the scenario where it is below zero. An edge-case where positions are being liquidated with a leftover tick which is not liquidated and at the same time in bad debt such that balanceLong is negative will result in an erroneous reset of balanceLong to zero and the subtraction of the negative value from balanceVault. Once the position is then liquidated, balanceVault will be decreased and balanceLong increased. This issue is only applicable **if there are for example 3 bad-debt positions and only one is liquidated**. If there are two and one is liquidated, it would be as follows: balanceLong = -10e18 balanceVault = 100e18 a) _handleNegativeBalances is invoked balanceLong = 0e18 balanceVault = 90e18 One would assume that: b) position gets liquidated balanceLong = 10e18 balanceVault = 80e18 However, due to the balanceLong set to zero, the position value of that position gets adjusted again.

Therefore, for this issue to occur, there would need to be at least two unliquidated bad-debt ticks, where one of them would obtain the negative value and one obtain the positive value.

PoC:

```
function testLongBalanceClampLeadsToExcessGainLongSide() public {
        // in case of negative long balance amount the balance is reset to 0 which makes the leftover positive position gain more assets
        vm.prank(ADMIN);
        protocol.setFundingSF(0);

        (Position memory position,) = protocol.getLongPosition(initialPosition);
        console.log("initialPositionExposure %18e", position.totalExpo);
        console.log("initialPositionAmount %18e", position.amount);

        // leverage approx 10x
        (, PositionId memory posId) = protocol.initiateOpenPosition(
        uint128(LONG_AMOUNT/2),
        CURRENT_PRICE * 9 / 10,
        type(uint128).max,
        protocol.getMaxLeverage(),
        address(this),
        payable(this),
        type(uint256).max,
        abi.encode(CURRENT_PRICE),
        EMPTY_PREVIOUS_DATA
        );

        (, PositionId memory posId2) = protocol.initiateOpenPosition(
        uint128(LONG_AMOUNT * 20),
        CURRENT_PRICE * 89 / 100,
        type(uint128).max,
        protocol.getMaxLeverage(),
```

```
        address(1234),
        payable(address(1234)),
        type(uint256).max,
        abi.encode(CURRENT_PRICE),
        EMPTY_PREVIOUS_DATA
        );

                {
        uint256 vaultbalance = protocol.getBalanceVault();
        console.log("vaultBalance %18e", vaultbalance);

        uint256 longBalance = protocol.getBalanceLong();
        console.log("longBalance %18e", longBalance);

        uint totalBalance = vaultbalance + longBalance;
        console.log("totalBalance %18e", totalBalance);
        }


        {
        uint128 initialPosLiqPrice =
protocol.getEffectivePriceForTick(initialPosition.tick);
        uint128 firstTickPrice =
protocol.getEffectivePriceForTick(posId.tick);
        uint128 secondTickPrice =
protocol.getEffectivePriceForTick(posId2.tick);

        console.log("initial position liquidation price %18e",
initialPosLiqPrice);
        console.log("first tick liquidation price %18e", firstTickPrice);
        console.log("second tick liquidation price %18e",
secondTickPrice);

        uint128 newPrice = 1e18;
```

	_waitDelay(); *// in case this validation is performed, the balance will be clamped leading to the issue. commenting this line out will yield the correct result of 0 longAmount left* *protocol.validateOpenPosition(payable(this), abi.encode(newPrice), EMPTY_PREVIOUS_DATA);* *console.log("newPrice",newPrice);* *skip(100);* *protocol.liquidate(abi.encode(newPrice));* *}* *{* *// there would be atleast one positions with non-zero balance* *uint256 vaultbalance = protocol.getBalanceVault();* *console.log("vaultBalance %18e", vaultbalance);* *uint256 longBalance = protocol.getBalanceLong();* *console.log("longBalance %18e", longBalance);* *uint totalBalance = vaultbalance + longBalance;* *console.log("totalBalance %18e", totalBalance);* *}* *}*
Recommendations	A solution for this issue is non-trivial as it means liquidations must forcefully liquidate all liquidatable ticks. This may lead to excessive gas consumption. An idea may be to force liquidate all positions (during the liquidate call and core update) while at the same time implementing a governance

	function which allows to liquidate single ticks in case the liquidation call would be too expensive. This function can also be accessible to different roles. Another solution might be the surveillance of the protocol, paired with position openings in the scenario where the protocol moves towards its natural limitation.
Comments / Resolution	Acknowledged. The Smardex team added the following comment: The primary concern associated with this issue is that long positions are liquidated at a stage where a substantial amount of bad debt may accumulate for the final positions within the protocol. However, given the minimum leverage requirement of 1.1x, an immediate 90% drop in the value of the underlying asset would be necessary to trigger such an outcome—an event that remains improbable. To mitigate potential risks and enhance the protocol resilience, we have implemented several measures, including: - Imbalance limits, which help maintain protocol stability by restricting excessive exposure, - Dedicated liquidation infrastructure, designed to execute liquidations even in scenarios where it is not immediately profitable, - Rebalancer (aka: Drip Accumulator) triggers - A commitment to consider opening positions if the long balance reaches critically low levels to ensure sustainable protocol operations.

Issue_32	Extrapolation within opening initiation can result in incorrect tick for position
Severity	**Medium**
Description	Whenever a new position is being opened, tradingExpo is extrapolated as follows: tradingExpo: Core.longTradingExpoWithFunding(lastPrice, uint128(block.timestamp)), This can become specifically a problem because the extrapolation assumes a hypothetical state and thus the tick for the corresponding position can be different as it should be.
Recommendations	Consider carefully documenting this issue.
Comments / Resolution	Acknowledged.

Issue_33	maxLeverage excess within _calcRebalancerPositionTick
Severity	**Medium**
Description	Within the _calcRebalancerPositionTick function, the corresponding tick for the desired leverage (tradingExpo) and current price is calculated. It is possible that the calculated leverage changes and exceeds the maximum leverage. This can happen in the scenario where the current tick has a different penalty than the protocol tick. While the position will be stored at that tick, in case the penalty is now smaller than expected, the leverage will be increased because liqPriceWithoutPenalty becomes larger: *if (posData_.liquidationPenalty != data.currentLiqPenalty) {* *data.liqPriceWithoutPenalty = Utils._getEffectivePriceForTick(* *Utils._calcTickWithoutPenalty(posData_.tick, posData_.liquidationPenalty),* *lastPrice,* *cache.tradingExpo,* *cache.liqMultiplierAccumulator* *);* *}* This will furthermore impact the tradingExpo, since totalExpo will now be larger. In a similar fashion, if the penalty is larger, totalExpo will become smaller.
Recommendations	Consider documenting this issue.
Comments / Resolution	Acknowledged.

Issue_34	Edge-case can result in lower leverage than expected
Severity	**Medium**
Description	Within the _calcRebalancerPositionTick function, the following special case is introduced: *// due to the rounding down, if the imbalance is still greater than the desired imbalance* *// and the position is not at the max leverage, add one tick* *if (* *data.highestUsableTradingExpo != tradingExpoToFill* *&& Utils._calcImbalanceCloseBps(* *cache.vaultBalance.toInt256(),* *(cache.longBalance + positionAmount).toInt256(),* *cache.totalExpo + posData_.totalExpo* *) > data.longImbalanceTargetBps* *) {* *posData_.tick += s._tickSpacing;* *posData_.liquidationPenalty = getTickLiquidationPenalty(posData_.tick);* *data.liqPriceWithoutPenalty = Utils._getEffectivePriceForTick(* *Utils._calcTickWithoutPenalty(posData_.tick, posData_.liquidationPenalty),* *lastPrice,* *cache.tradingExpo,* *cache.liqMultiplierAccumulator* *);* *posData_.totalExpo = Utils._calcPositionTotalExpo(positionAmount, lastPrice, data.liqPriceWithoutPenalty);* *}* This case is to ensure that the highest possible expo is still met, even if the rounding down in the tick determination results in not meeting it.

	In the special scenario where the liquidationPenalty of that tick is higher than the current liquidation penalty, liqPriceWithoutPenalty will result in a smaller value than expected, which then can in fact result in a lower leverage compared to the scenario where the function does not enter into the if-clause. Contrary to the previous issue, this is an avoidable scenario where simply the previously calculated tick could have been used.
Recommendations	Consider checking which tick offers the more favorable price to meet the targetTradingExpo.
Comments / Resolution	Acknowledged.

Issue_35	Extrapolation can result in inaccurate posId.tick
Severity	**Medium**
Description	Within the _prepareInitiateOpenPositionData function, the tradingExpo is calculated using extrapolation: *tradingExpo: Core.longTradingExpoWithFunding(lastPrice, uint128(block.timestamp)),* This can result in inaccurate unadjustment which does not reflect the real unadjustment scenario (in case where the funding is different). This can then furthermore result in an unexpected tick where the position is stored.
Recommendations	Consider implementing a slippage check for the tick where the position is stored.

Comments / Resolution	Acknowledged.

Issue_36	Rebalancer leverage may often be insufficient to reach target
Severity	**Medium**
Description	As by design, the rebalancer is not allowed to open a position with a leverage more than the leverage limitation. This will inherently result in scenarios where the neede tradingExpoToFill is not met and thus the protocol cannot be rebalanced.
Recommendations	Consider if it is desirable to allow the rebalancer exceeding maxLeverage in case it is needed.
Comments / Resolution	Acknowledged.

Issue_37	Casting within _handleNegativeBalances can revert in edge-case
Severity	**Medium**
Description	The _handleNegativeBalances function is responsible for setting negative balances to zero and applying the delta on the counter value. If however, under any specific edge-case it happens that both parameters are negative or the negative nominal value is larger than the counter parameter's positive value. E.g. balanceVault = 0.5; balanceLong = -0.6 Then the toUint256 cast will revert. Since the value transitions **could** experience extreme edge-cases, such as: - Bad debt scenario - Large funding fee (protocol fee) - .... **One definitive scenario would be the case where the protocol fee is larger than the aggregated values:** Long = 0.01 Vault = 0.1 Funding = 1.5 (based on past EMA where long was in favor); protocolFee = 10% Long = -1.49 Vault = 1.35 There may be multiple other scenarios where this can become an issue. However, since the fix remains rather trivial and does not introduce obvious side-effects, we allocate our resources to other

	parts of the code instead of further investigating edge-cases where this could become a problem.
Recommendations	Consider simply clamping any remaining negative value to zero before the uint256 cast. Furthermore, in the scenario where the funding fee is larger, it should be kept in mind that the protocol now has insufficient assets to pay out the fee, this can be simply countered by transferring the missing assets manually to the contract which then still allows the protocol to continue working.
Comments / Resolution	Resolved.

Issue_38	Position value can never become worth more than position expo even with funding application
Severity	**Medium**
Description	The _positionValue function calculates the nominal value of a position as follows: positionTotalExpo * (currentPrice - liqPriceWithoutPenalty) / currentPrice The calculation in itself is correct. A problem just arises because liqPriceWithoutPenalty can become a minimum of zero, which is determined within the _adjustPrice function as follows: liqPriceNoPenaltyUnadjusted * (assetPrice*longTradingExpo) / accumulator If a user creates a position with minimum leverage and this funding is negative for a certain period of time, the result of the above formula will be zero, which effectively limits the position value growth up to the position expo. *This issue has been rated as medium severity after clarification with the Smardex team that this is a design choice. However, it is still mandatory to mention that in our opinion, a position should always gain value from a negative funding rate, which is simply not given due to the math design. *Further edge-cases may occur due to that design choice.*
Recommendations	A solution for this issue requires the liquidation price to become negative since the position value calculation is based on the liquidation price. Thus the math must be rewritten. Side-effects on the overall math logic must be considered and are very likely.

Comments / Resolution	Acknowledged, this is a design choice by the Smardex team.

Issue_39	Users can frontrun the oracle update and trigger the Rebalancer to immediately create a profitable position
Severity	**Medium**
Description	Throughout the architecture and most specifically for the creation of long positions, an initiate -> validate approach is used to prevent oracle update frontruns. If that approach would not be existent, users could inspect unupdated oracle prices, open a long position and then immediately profit from that opened long position. While the Rebalancer itself has an initiate -> validate approach as well, this approach has nothing to do with opening positions, in fact, once users have deposited into the rebalancer, a position is not opened immediately. If a user has deposited into the rebalancer and this deposit has been validated, it will now be used to open long positions via the _triggerRebalance function. Now we need to understand that this position opening is not subject to any fee or any down-sized price such as the standard init and valid price fetching mechanisms, instead it will simply use the _lastPrice, which is the neutral price returned from the oracle. This therefore allows the user to search for a slightly outdated oracle price which is lower than the real current price, making the position immediately profitable. In reverse, a malicious user can also fetch a price which is higher than

	the current price, resulting in an immediate loss for rebalancer users.
Recommendations	Consider always using the updated price for rebalancer position openings.
Comments / Resolution	Acknowledged.

Issue_40	minLongPositionCheck does not include bonus
Severity	**Medium**
Description	Within the _triggerRebalancer function, the following check is executed: *// if the amount in the position we wanted to open is below a fraction of the _minLongPosition setting,* *// we are dealing with dust. So we should stop the process and gift the remaining value to the vault* *if (data.positionAmount <= s._minLongPosition / 10_000) {* *// make the rebalancer believe that the previous position was liquidated,* *// and inform it that no new position was open so it can start anew* *rebalancer.updatePosition(Types.PositionId(Constants.NO_POSITION_TICK, 0, 0), 0);* *vaultBalance_ += data.positionValue;* *return (longBalance_, vaultBalance_, Types.RebalancerAction.Closed);* *}* The problem with this check is that the potential bonus is not incorporated, while in fact this bonus should flow into the new

	rebalancer position. It is important to ensure if the positionAmount is zero that the position should not be opened as this means the bonus would effectively be wasted because there is no party on the rebalancer to receive it.
Recommendations	Consider incorporating the bonus in this check while ensuring that positionAmount must not be zero.
Comments / Resolution	Failed resolution, a new high risk issue has been introduced. We recommend reverting the implementation to its previous version acknowledging this issue in an effort to prevent the introduction of new side effects. **Resolution 2:** Acknowledged. The _triggerRebalancer function has been adjusted which acknowledges this issue.

Issue_41	Erroneous imbalance check due to incorporation of bonus can leave protocol imbalanced
Severity	**Medium**
Description	The _triggerRebalancer function executes a check to ensure the rebalancer is only triggered whenever the protocol is imbalanced: *int256 currentImbalance = Utils._calcImbalanceCloseBps(* *cache.vaultBalance.toInt256(),* *cache.longBalance.toInt256(), cache.totalExpo* *);* *// if the imbalance is lower than the threshold, return* *if (currentImbalance <= s._closeExpoImbalanceLimitBps) {* *return (longBalance_, vaultBalance_,* *Types.RebalancerAction.NoImbalance);* *}* *}* The problem with this check is that the bonus is already deducted from the vault side: cache.vaultBalance -= bonus; In the scenario where the imbalance check is considered as “ok”, the protocol still may remain imbalanced because the bonus is not included. In reality, the bonus should not be incorporated into the imbalance check. It is important to keep in mind that an adjustment of the initial imbalance check will then eventually mess with the following check: *// make sure that the rebalancer was not triggered without a*

	sufficient imbalance *// as we check the imbalance above, this should not happen* *if (cache.tradingExpo >= targetTradingExpo) {* *revert IUsdnProtocolErrors.UsdnProtocolInvalidRebalancerTick();* *}* Especially, if the close and target imbalance does not deviate too much. Therefore, this issue must either be acknowledged or a proper refactoring of the whole flow must be implemented. Additionally, if we would follow the false check, then we would need to apply the bonus towards the long side as well and not only deducting it from the vault. However, the general concept of this practice is simply incorrect.
Recommendations	Consider refactoring this process or acknowledging this issue and accept this side-effect.
Comments / Resolution	Acknowledged.

Issue_42	Payout of liquidation rewards post rebalancing may put protocol below imbalance target
Severity	**Medium**
Description	The liquidation reward is paid out once the protocol has been rebalanced. As the payout will reduce _balanceVault this will make the rebalance process slightly void (depending on the payout amount), because the protocol may now deviate from the target imbalance. Furthermore, the USDN rebase may now be erroneously triggered as _balanceVault is now smaller.
Recommendations	There is no simple fix as the liquidation reward calculation relies on the rebalancer execution to properly determine the reward amount.
Comments / Resolution	Acknowledged.

Issue_43	Funding is erroneously impacted by temporare bad-debt scenario
Severity	**Medium**
Description	Whenever the price becomes higher than the liquidation price of a position, the position value becomes effectively negative which is temporarily reflected in a decrease of _balanceLong and increase of _balanceVault. Once a liquidation happens, this temporare state is reversed again. However, in the meantime this faulty state will impact the funding rate erroneously.
Recommendations	There is no trivial fix for this issue.
Comments / Resolution	Acknowledged.

Issue_44	Rebalancer is only triggered if there are no pending liquidations left
Severity	**Medium**
Description	As by design, the rebalance is only triggered if there are no pending liquidations left: *if (!isLiquidationPending_ && liquidationEffects.liquidatedTicks.length > 0) {* While it is prevented that further initiations/validations are happening as long as liquidations are pending, the funding rate is impacted by one side being excessive. A malicious user could then forcefully only liquidate one tick at a time to bring the protocol in an imbalanced state while preventing rebalancing. Furthermore, the bonus is deducted from the vault side in the scenario where the rebalancer is triggered. That means a user who is invested in the vault side can set up a liquidation bot which only liquidates one tick at a time, resulting in leftover liquidations and thus most of the time preventing the payout of the bonus such that the vault remains this as balance.
Recommendations	A solution for this issue would be to trigger the rebalancer upon every liquidation. However, this may introduce other side-effects.
Comments / Resolution	Acknowledged.

Issue_45	Bonus is incorrectly impacted by bad debt scenario
Severity	**Medium**
Description	The bad debt is a scenario where a position value temporarily becomes negative and this temporary PnL is transferred to the vault side. Once such a position is liquidated, this value is just transferred back from the vault to the long side, as if the position would have been liquidated on time without any remaining collateral (with the only side-effect that funding was impacted in the time-being). This means that the actual liquidated position should correspond with remainingCollateral = 0 and not with negative collateral for the bonus context. This is however incorrectly handled as in fact the negative collateral is deducted from the remaining collateral. As mentioned, in itself it is correct for the vault <-> long transfer. However, remainingCollateral is also used to calculate the bonus which now means this bad-debt scenario incorrectly decreases the bonus.
Recommendations	Consider refactoring the function to not deduct the bad-debt from the bonus.
Comments / Resolution	Acknowledged.

SettersLibrary

The ProtocolSetters library is invoked by the UsdnProtocol contract and incorporates the logic for parameter changes.

Issue_46	Governance Privilege: Change of critical parameters
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, governance can change the _oracleMiddleware contract which then allows for setting a dummy oracle to steal funds from the contract or the _liquidationRewardsManager to steal funds via a liquidation reward. There are several other possible malicious actions which can result in a loss of funds or stuck funds.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_47	Hindsight changes of parameter settings
Severity	**Medium**
Description	Several parameter changes will impact the protocol state in hindsight: - _rebalancerBonusBps - _protocolFeeBps - EMAPeriod This will have unexpected effects on the protocol state as for example the protocol will now apply the new _protocolFeeBps also on a period which was before the update.
Recommendations	Consider updating the protocol state beforehand
Comments / Resolution	Acknowledged.

Issue_48	Change of _liquidationPenalty can result in unexpected leverage change for users
Severity	**Medium**
Description	The _liquidationPenalty represents the value that is applied on the nominal liquidation price. This is the final value at which a position is considered as liquidatable. Since this value can be changed, one could think that it can accidentally happen that users accept a larger penalty than expected, this is partially the case but not the case in the scenario where the leverage is near the maximum. The reason for that is that users will provide the desiredLiqPrice parameter which already includes the penalty, so in the scenario where the penalty is increased and the position should be using the maximum leverage, the leverage safeguard is triggered and the call would actually revert. A different scenario applies if there are initiated positions and the penalty is changed during this time, if now the maxLeverage scenario is triggered, these positions will be potentially stored with the new penalty which means that the real leverage will be different from the desired leverage
Recommendations	Consider executing a check that there are no initiated long positions before the _liquidationPenalty is changed.
Comments / Resolution	Acknowledged.

UtilsLibrary

The UtilsLibrary contract exposes several helper functions which are used throughout the architecture. These are notably:

- Ether refunds
- Fee processing
- Oracle price fetching
- PnL helper calculation
- TickHash calculation
- Pending conversions
- Position value calculation
- Expo calculation
- Transfer callbacks
- Adjustments/Unadjustments

No issues found.

VaultLibrary

The VaultLibrary exposes the whole logic for depositing and withdrawing from the vault including all initiate and validation flows. Furthermore, it handles the logic for executing actionable pending actions which is tied to any initiation and validation.

The core idea behind deposits and withdrawals is to account for PnL in case it is in favor of the vault while ensuring that deposits do not impact the funding rate nor experience funding.

Issue_49	Incorrect setting of initiate data due to extrapolation
Severity	**High**
Description	For deposits and withdrawals, the core state is extrapolated. See the following example: *data_.balanceVault = vaultAssetAvailableWithFunding(data_.lastPrice, uint128(block.timestamp))* There are two problems with this extrapolation: a) The protocolFee is always ignored which means that balanceVault is always larger than it should be. This is especially a problem for withdrawals as users will then gain more wstETH than anticipated. b) The extrapolation is not reflected on balanceLong. In the deposit scenario this means the PnL will be applied based on a tradingExpo which does not incorporate the extrapolated balanceLong and will thus reflect an incorrect PnL (for the validation step). Furthermore, this will falsify the usdnPrice (which is a non-issue as its only informational).
Recommendations	Consider properly incorporating the _protcolFee as well as extrapolating balanceLong to then correctly calculate the PnL within the validation based on the corrected balanceLong value.
Comments / Resolution	Resolved.

Issue_50	SDEX burn towards dead address can be bypassed if future protocols implement a burn
Severity	**Medium**
Description	Currently, within the deposit function a fee in SDEX is being transferred from the user to the burn address upon the callback. This is problematic as Smardex has/will launch future protocols where SDEX is being burned and this can then be abused by reentering into one of these protocols. For example, UsdnLongFarming.harvest with a liquidated position can be called which transfers tokens to the dead address. (This contract is in a different scope)
Recommendations	Consider transferring SDEX manually out from the USDN protocol to the burn address.
Comments / Resolution	Resolved.

Issue_51	Extrapolation can be different from real funding
Severity	**Medium**
Description	During deposits and withdrawals (and also in other functionalities within the architecture), the funding rate is extrapolated up to the current block.timestamp. In some extreme scenarios, this extrapolation can be completely incorrect, if for example after the extrapolation a large withdrawal is executed with the oracle return timestamp still being in the past. This would then manipulate the funding which can potentially go even in the opposite direction than the assumed extrapolation.
Recommendations	We do not recommend a change as the removal of extrapolation will re-introduce other issues. However, this issue should be kept in mind.
Comments / Resolution	Acknowledged.

Issue_52	Pending vault validations will impact the vault state
Severity	**Medium**
Description	Whenever a deposit/withdrawal is initiated, the ratio which is used is determined during the initiated state. The only thing which changes during the validation state is any eventual price change. This can become a problem if a vault validation is pending for some time and now the actual vault ratio has changed, as the validated action now will flow into the new vault ratio and manipulate it, which can be in both directions (to the favor or against the favor of the vault). This opens the possibility for MEV.

	The same can be applied for late position closures which impact the vault state (increase/decrease share value). Furthermore, an even more intrusive issue occurs if a withdrawal was initiated during a timestamp where shares were very valuable, then until the validation the balanceVault decreases significantly. This can end up in such a situation where the initiators to be received wstETH amount exceeds balanceVault which leaves other shareholders left with nothing.
Recommendations	A fix for this issue means that the whole deposit and withdrawal logic needs to be refactored.
Comments / Resolution	Acknowledged.

Issue_53	Several scenarios allow for frontrunning interactions
Severity	**Medium**
Description	Upon deposit (and withdrawals), the initial state is cached and then used for the validation later. This means any change in balanceVault between the initiate and validate timestamp is in favor of the depositor, as the depositor could immediately withdraw again with a profit. Since liquidations are increasing balanceVault (in a non-bad-debt scenario), this means depositors will benefit from it. Now it is theoretically possible that a depositor frontruns a liquidation. However, practically that is slightly harder to execute as the protocol does not allow for initiations/validations if there are any pending liquidations. However, since it is possible for the user to: a) Choose the latest pyth price which has been updated b) Predict a state where the next tick below the currentPrice has a large value It is possible for a user to extract value from the protocol under specific circumstances. The same can be applied vice-versa for withdrawals. Furthermore, in the scenario of a late position closure, the PnL is applied and thus balanceVault increased/decreased: *if (assetToTransfer < long.closeBoundedPositionValue) {* *uint256 remainingCollateral;* *unchecked {* *// since assetToTransfer is strictly smaller than closeBoundedPositionValue,* *// this operation can't underflow* *remainingCollateral = long.closeBoundedPositionValue -*

```
assetToTransfer;
                }
                s._balanceVault += remainingCollateral;
            } else if (assetToTransfer > long.closeBoundedPositionValue) {
                uint256 missingValue;
                unchecked {
                // since assetToTransfer is strictly larger than
closeBoundedPositionValue,
                // this operation can't underflow
                missingValue = assetToTransfer -
long.closeBoundedPositionValue;
                }
                uint256 balanceVault = s._balanceVault;
                // if the vault does not have enough balance left to pay
out the missing value, we take what we can
                if (missingValue > balanceVault) {
                s._balanceVault = 0;
                unchecked {
                // since `missingValue` is strictly larger than `balanceVault`,
                // their subtraction can't underflow
                // moreover, since (missingValue - balanceVault) is smaller
than or equal to `missingValue`,
                // and since `missingValue` is smaller than or equal to
`assetToTransfer`,
                // (missingValue - balanceVault) is smaller than or equal to
`assetToTransfer`,
                // and their subtraction can't underflow
                assetToTransfer -= missingValue - balanceVault;
                }
                } else {
                unchecked {
                // as `missingValue` is smaller than or equal to
`balanceVault`, this operation can't underflow
                s._balanceVault = balanceVault - missingValue;
                }
```

	} This can be frontrun with either a deposit (in case balanceVault increases) or frontrun with a withdrawal (in case balanceVault decreases). Moreover, any funding which is applied to the protocol b/w [initiate;validate] will result in changing the protocol state compared to the initiate state, resulting in a outcome for deposits/withdrawals will may result in an immediate gain/loss for users.
Recommendations	A fix for this issue means that the whole deposit and withdrawal logic needs to be refactored.
Comments / Resolution	Acknowledged.

Issue_54	Funding adjustment can be blocked due to _pendingBalanceVault mechanism
Severity	**Medium**
Description	The _pendingBalanceVault variable is increased/decreased whenever a deposit or withdrawal has been initiated. It flows into imbalance checks as it now acts as if the deposit/withdrawal has already happened. A problem from this implementation is the fact that this does not actually affect the funding. In itself, this logic (to not affect the funding) is totally correct as these actions have not yet been validated. A problem is however the fact that (for example) if the protocol is imbalanced towards the long side, the funding is considered as positive. A user can now initiate a deposit which brings the protocol into balance again but this deposit does not impact the funding. The protocol is then in a state where it is actually not considered as imbalanced but the funding is still positive. A user can abuse this mechanism to keep an artificially high funding rate while the protocol is imbalanced towards the vault side (by initiating deposits), which means position closures (which would actually decrease the funding rate) are not allowed. This can be done for up to 15 minutes as after then, anyone can validate a deposit.
Recommendations	Consider wrapping ETH to WETH in case the validator rejects native ETH transfers. This will ensure this issue cannot be exploited for up to 15 minutes.
Comments / Resolution	Acknowledged.

Issue_55	Incorrect imbalance check during deposits
Severity	**Medium**
Description	The current imbalance check for the deposit initiation is happening based on the non-extrapolated state, which means if the timestamp returned by the oracle is < block.timestamp, the imbalance check is simply inaccurate. This issue is present throughout the codebase but highlighted here as the deposit initiation is one dedicated spot where the state is extrapolated.
Recommendations	Consider incorporating the extrapolation into the imbalance check.
Comments / Resolution	Acknowledged.

Issue_56	Missing slippage check for burned SDEX amount
Severity	**Medium**
Description	Currently, the burned SDEX amount is calculated based on the preliminary received USDN token amount: *uint256 usdnToMintEstimated = usdn.convertToTokens(usdnSharesToMintEstimated);* *// we want to at least mint 1 wei of USDN* *if (usdnToMintEstimated == 0) {* *revert IUsdnProtocolErrors.UsdnProtocolDepositTooSmall();* *}* *uint32 burnRatio = s._sdexBurnOnDepositRatio;* *data_.sdexToBurn = Utils._calcSdexToBurn(usdnToMintEstimated, burnRatio);*

	Since the USDN token amount can fluctuate based on the USDN price and has nothing to do with the actual amount of shares which are received (besides the conversion process), it can happen that more SDEX tokens than expected are burned from a user.
Recommendations	Consider incorporating a slippage check,
Comments / Resolution	Acknowledged.

UsdnProtocol

Impl

The Impl contract is part of the entry architecture which allows users to interact with the USDN protocol. It is responsible to initialize the contract storage and contains logic for the UUPS upgrade as well as delegates calls with no matching function selector to the Fallback address. No issues found.

Actions

The Actions contract is part of the entry architecture which allows users to interact with the USDN protocol to initiate and validate actions, liquidate positions, transfer the position ownership as well as validate actionable actions.

Furthermore it uses the domainSeparatorV4 for signature integrity.

No issues found.

Core

The Core contract is part of the entry architecture which allows users to interact with the USDN protocol. It is solely responsible for initializing the protocol by the DEFAULT_ADMIN.

No issues found.

Long

The Long contract is part of the entry architecture which allows users to interact with the USDN protocol. It exposes view-only functions which are related to long positions.

No issues found.

Vault

The Vault contract is part of the entry architecture which allows users to interact with the USDN protocol. It exposes view-only function to return the current USDN price as well as the vault balance including extrapolated funding.

No issues found.

Fallback

The Fallback contract is part of the entry architecture which allows users to interact with the USDN protocol. It is deployed as a standalone, external contract and is invoked by the main entry architecture whenever the fallback function is triggered which means that none of the main function selectors is triggered.

It acts as an implementation contract while still using the main protocol's storage.

This contract mainly exposes view-only functions and governed functionalities to alter the protocols settings.

Issue_57	Pausing does not settle funding
Severity	**Medium**
Description	Currently, the pause function is meant to update the funding up to the current block.timestamp: *Core._applyPnlAndFunding(Utils._getMainStorage()._lastPrice, uint128(block.timestamp));* This is however not written to storage as the temp data is only cached into memory.
Recommendations	Consider writing the change into storage as well as accruing the protocol fee.
Comments / Resolution	Resolved.

Resolution Issues

Issue_58	Edge-case within _triggerRebalancer will result in rebalancer opening with no corresponding processed user deposit and incorrect bonus application
Severity	**High**
Description	The _triggerRebalancer function has been refactored against the recommendation from Bailsec. This refactoring has introduced a high risk issue which results in a loss of the bonus and a corresponding faulty rebalancer position opening. This happens in a specific edge-case with the following conditions: - bonus != 0 - _pendingAssetsAmount = 0 - data.prevPositionAmount != 0 - realPositionValue < data.positionAmount - entryAccMultiplier near 10_000 To explain the issue, we need to follow the call-path of the _triggerRebalancer function with these conditions: a) Protocol is considered as imbalanced b) Old position is closed and realPositionValue is now a smaller value than data.prevPositionAmount. This will have the effect that accMultiplier decreases (see subsequent steps) c) newAccMultiplier is now calculated and becomes < 10_000 d) bonus + positionAmount is larger than _minLongPosition/ 10_000 which bypasses that condition e) Since data.prevPositionAmount > 0 and the new multiplier < 10_000, it will automatically give the realPositionValue to the vault and deduct data.positionValue from data.positionAmount and sets data.positionValue to zero

	f) This means data.positionAmount as well as data.positionValue is zero, resulting in a zero value transfer of wstETH g) However, the bonus is non-zero which means the bonus is deducted from balanceVault and added to positionAmount h) The bonus (data.positionAmount) is added to the long side i) A new position is now opened and the old positionVersion in the vault is liquidated. At the same time, since _pendingAssetsAmount is zero, which means there are no to-processed deposits in the rebalancer, the rebalancer opens a long position which only includes the bonus that no one can claim, which is a highly faulty state
Recommendations	Consider reverting all changes made to the _triggerRebalancer flow.
Comments / Resolution	Resolved.