BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Smardex

USDN

May 2024

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
Website	smardex.io
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/SmarDex-Ecosystem/usdn-contracts/tree/ff2b96fefc6ab41bb2adccef430c1f2f19e29dda
Resolution 1	A resolution round has not been included in this report, as a second full audit has been concluded for this scope.

2. Detection Overview

A resolution round has not been included in this report, as a second full audit has been concluded for this scope.

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	31			
Medium	21			1
Low	29	2		
Informational	18			
Governance	3			
Total	102	2		1

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

Note: The report may mention ETH instead of wStETH. However, most of the time the reference is meant to be wStETH for the vault asset and oracle price.

The USDN protocol which was developed by SmarDex's engineering team is a hybrid vault and perpetual futures protocol where users can deposit wStETH and receive USDN as well as long wStETH with leverage by providing wStETH which serves as collateral including the cross-margin option.

Whenever users deposit wStETH, they will receive the corresponding value of USDN based on the total wStETH value on the vault side of the protocol and the total USDN supply. The USDN token serves as deposit receipt for the staking position but will also have a large variety of use-cases in the SmarDex and overall DeFi ecosystem. Users can redeem their USDN tokens at any time back to wStETH, following the already explained conversion process. **This protocol is not a pure stablecoin protocol:** The USDN coin is meant to stay in a stable price range between 1.0087 and 1.009 due to many different mechanisms, most importantly the rebase mechanism. This will however only work if the ETH price stays consistent or increases, which is owed to the fact that USDN is **solely backed by wstETH.**

In such a scenario where the ETH price drops, this means the nominal USD value in the vault will drop as well and while users will still receive the same wstETH amount, the value of this wstETH is less in terms of USD

Whenever users long ETH, they will deposit wStETH as collateral and can then bet on an increasing ETH price. This concept follows the widely known "cross-margin" concept where the already deposited ETH serves as collateral to purchase perpetual future contracts.

The most interesting part of the protocol is the value transfer between the vault and the long side. Below follows a quick summary but we will cover each mechanism in-depth in each corresponding report section.

a) Funding fee: The rationale behind this fee is to keep the long exposure and the vault exposure in equilibrium. If the vault exposure dominates, this will decrease the funding fee, incentivizing users to long wstETH as the vault side will pay funding to long positions.

In reverse, if the long exposure dominates, users will pay a funding fee on their long positions which is being transferred to the vault side.

b) Pnl: Whenever long positions are opened, this means users will gain a profit or loss depending on the price movements and the used leverage. Any loss will increase the value on the vault side whereas any profit will be taken from the vault side.

c) Liquidations: There is a saying *"Liquidations are a forced transfer of wealth from leveraged future traders to wealthy spot buyers"*. This saying is perfectly aligned with Smardex's strategy, as any liquidation will increase the value on the vault side.

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.

Privileged Functions

- none

Issue_01	Contract contains redundant logic
Severity	**Informational**
Description	Currently, the contract contains several spots which are redundant and never used. For example the pushFront function is never used in the context of queue additions or the atRaw function contains the following unreachable code: *if (deque._begin > deque._end) {* *// here the values are split at the beginning and end of the range, so invalid indices are in the middle* *if (rawIndex < deque._begin && rawIndex >= deque._end) revert QueueOutOfBounds();* *}* This makes the contract less readable and more complex for third-party inspectors.
Recommendations	Currently, we do not recommend a change. However, it can be explored if these redundant parts shall be removed in the future.
Comments / Resolution	

HugeUint

The HugeUint library is a library that allows arithmetic operations on uint512 data types. Since solidity does only support up to uint256 (2**256-1), in some situations there is the need for larger data types. Usually, this may be the case for large financial amounts or some cryptographic operations.

In this architecture, the only use-case is the _liqMultiplierAccumulator, since this variable may become larger than uint256.

The way this library works is by exposing a uint512 struct which includes the low 256 bits (lo) and the high 256 bits (hi).

Privileged Functions

- none

No issues found.

Permit2TokenBitfield

The Permit2TokenBitfield library leverages bitwise operations to check if a flag is set. More specifically, it checks based on the provided Bitfield, if the permit2 solution should be used for wStETH and for SDEX.

On a meta-level description, users can just provide “1” (00000001) if they want to use permit2 for wStETH only, “2” (00000010) for SDEX only and “3” (00000011) for both tokens.

This library is solely used within the ActionsVault and the ActionsLong contracts to determine if tokens are transferred via standard transferFrom or via Uniswaps permit2 architecture. More information on this topic can be found here:

https://blog.uniswap.org/permit2-and-universal-router

Privileged Functions

- none

SignedMath

The SignedMath library is a simple library that handles addition, subtraction, multiplication and division while enforcing overflow checks. It is used throughout the architecture.

Privileged Functions

- none

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.

Privileged Functions

- none

Issue_02	The getTickAtPrice function doesn't return the largest tick whose price is less than or equal to the given price
Severity	**Informational**
Description	Currently, the getTickAtPrice function doesn't return the largest tick whose price is less than or equal to the given price. It returns a value that is always close to the actual tick. From our testing, it is always between a +-1 range to the actual value. Since this function is used to determine the tick where a position is stored, it will result in users having a lower liquidation price which means a position is potentially liquidated later than expected
Recommendations	Consider updating the getTickAtPrice function to return the actual largest tick whose price is less than or equal to the given price.
Comments / Resolution	

OracleMiddleWare

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 protocol assumes the following gas spendings per operation:

a) General gas for the overall call: 432_860
b) Each tick liquidation: 55_890
c) Rebalancer: 251_476
d) Rebase: 3_447
e) Base cost: 21_000

Additionally, to incentivize users to liquidate ticks, the refund will expose a 3x multiplier on tick liquidations, which means users will receive 3x the gas spent for liquidations as refund.

For a full liquidation scenario with 1 tick, this will result in the following refund amount:

432860 + 21000 + 55890*3 + 3441 + 251476 = 876447

Currently, the gas price is 1250714246 WEI

Results in a refund amount of: 876447 * 1250714246 = 1096184748763962 (0.001096 ETH)

To counter griefing attacks, the protocol implements two safeguards which determine upper limits for the provided gas price:

a) Chainlink's Gas Oracle
b) Manual gasPriceLimit state variable

Privileged Functions

- transferOwnership
- renounceOwnership
- acceptOwnership
- setRewardsParameters

Issue_03	Chainlink gas price feed is deemed "unsuitable to use"
Severity	**High**
Description	As per Chainlink team, the Gas Oracle is considered as "unsuitable to use". Since we have no further information, we can only speculate about the meaning of this. In the worst case scenario, this means users could damage the protocol by artificially increasing the provided gasPrice, which then in turn drains the protocol because the upper limit safeguard from the Chainlink oracle is void. Fortunately, this can be countered by the gasPriceLimit safeguard. However, it has to be noted that the gasPriceLimit is currently set to 1000 GWEI. If we incorporate the 1000 GWEI value into our example from above, users would receive a refund of 0.876 ETH for a full liquidation flow. This value is sufficient to drain the protocol.

	(It is clear that users will not gain any significant value from this griefing attack)
Recommendations	Consider incorporating a reliable gas oracle and setting the gasPriceLimit to a reasonable value.
Comments / Resolution	

Issue_04	stETH depeg may prevent critical liquidations due to lack of incentives
Severity	**High**
Description	Since refunds are happening in wStETH instead of ETH, the contract uses the getWstETHByStETH function on the wStETH contract to determine how much wStETH are received for the ETH gas refund. This conversion process inherently assumes that the ETH <-> stETH conversion is 1:1, since the provided gas value was in ETH instead of stETH. In such a scenario where stETH depegs, users will receive the corresponding wstETH amount for stETH (instead of ETH). Illustrated this means that if a user provides 1 ETH, he will receive 0.85 wstETH. However, 0.85 wstETH corresponds to 1 stETH, which will then in turn be 0.95 ETH (if stETH is depegged). Thus liquidations may not be executed as planned by bots due to a lack of incentives and positions may become underwater.
Recommendations	Consider either using an oracle to convert the ETH to stETH price or manually increase the incentive in such situations.
Comments / Resolution	

Issue_05	Fast oracle safeguard may result in loss for liquidators
Severity	**Medium**
Description	**For this issue we assume that the oracle works as expected.** If now a user initiates a transaction with 5 GWEI and this transaction is stuck some time in the mempool until the gas price drops to let's say 2 GWEI, the user would effectively lose funds due to the upper safeguard of the fast oracle, as now the refund will be calculated based on 2 GWEI instead of 5 GWEI.
Recommendations	Consider implementing some wiggle-room in the fast oracle safeguard. Furthermore as we have already elaborated the refund amount should not be taken from the vault but a separate safety module.
Comments / Resolution	

OracleMiddleWare

The OracleMiddleWare contract is a modular oracle handler contract which is responsible for fetching the correct Oracle price for all interactions in the core protocol.

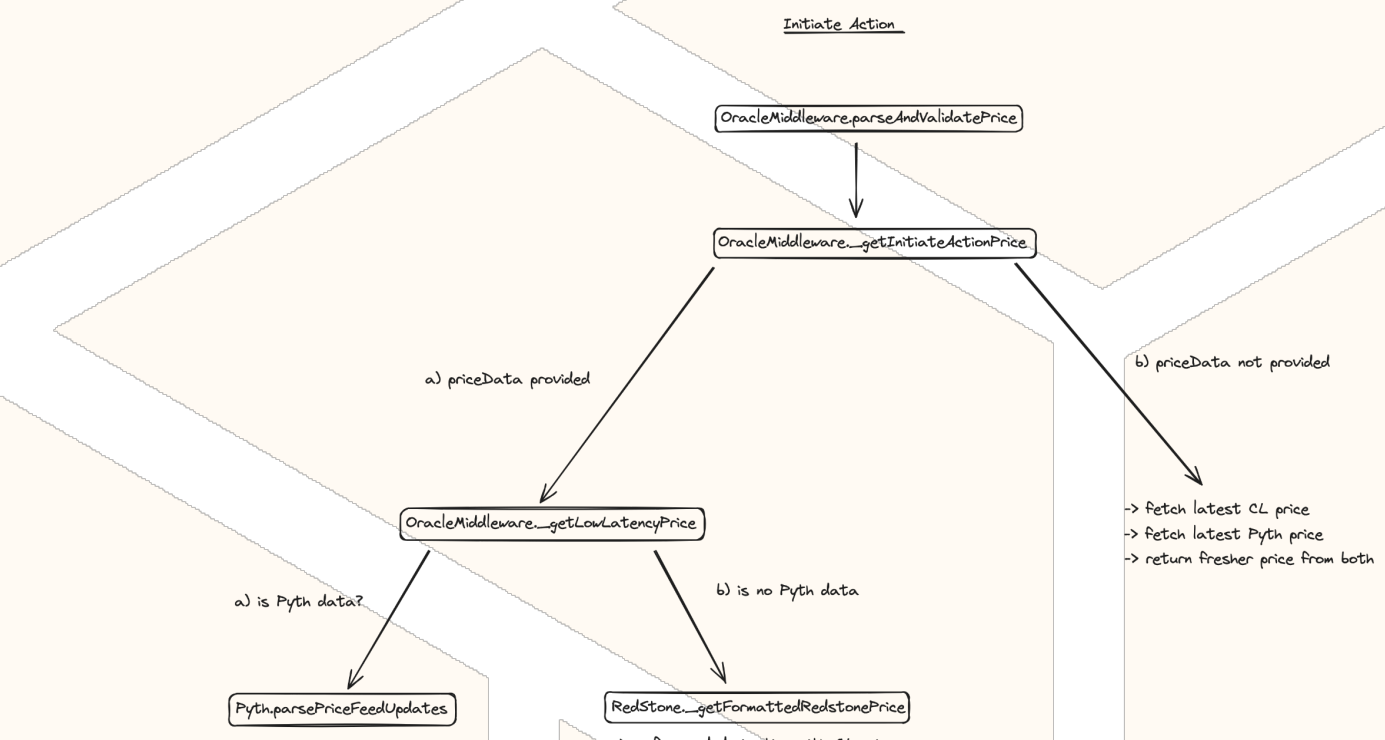
This contract evolves mainly around the parseAndValidatePrice function, which fetches prices from Chainlink, Pyth and Redstone, depending on the current circumstances and situation. It inherits all three oracle handler contracts (Chainlink, Pyth and Redstone).
It allows users to provide specific priceData which can either be priceData for the Redstone oracle or priceData for the Pyth oracle. In the scenario of a Pyth oracle, it will fetch the first 4 bytes of the priceData parameter and checks if this is 0x504e4155, which corresponds to PNAU (Pyth Network Accumulator Update). In the scenario where the Redstone oracle is meant to be used, the priceData parameter can be any arbitrary parameter as long as the first 4 bytes do not correspond to PNAU.

Due to a lack of Redstone documentation, the Redstone oracle implementation will be removed.

There are mainly two different scenarios:

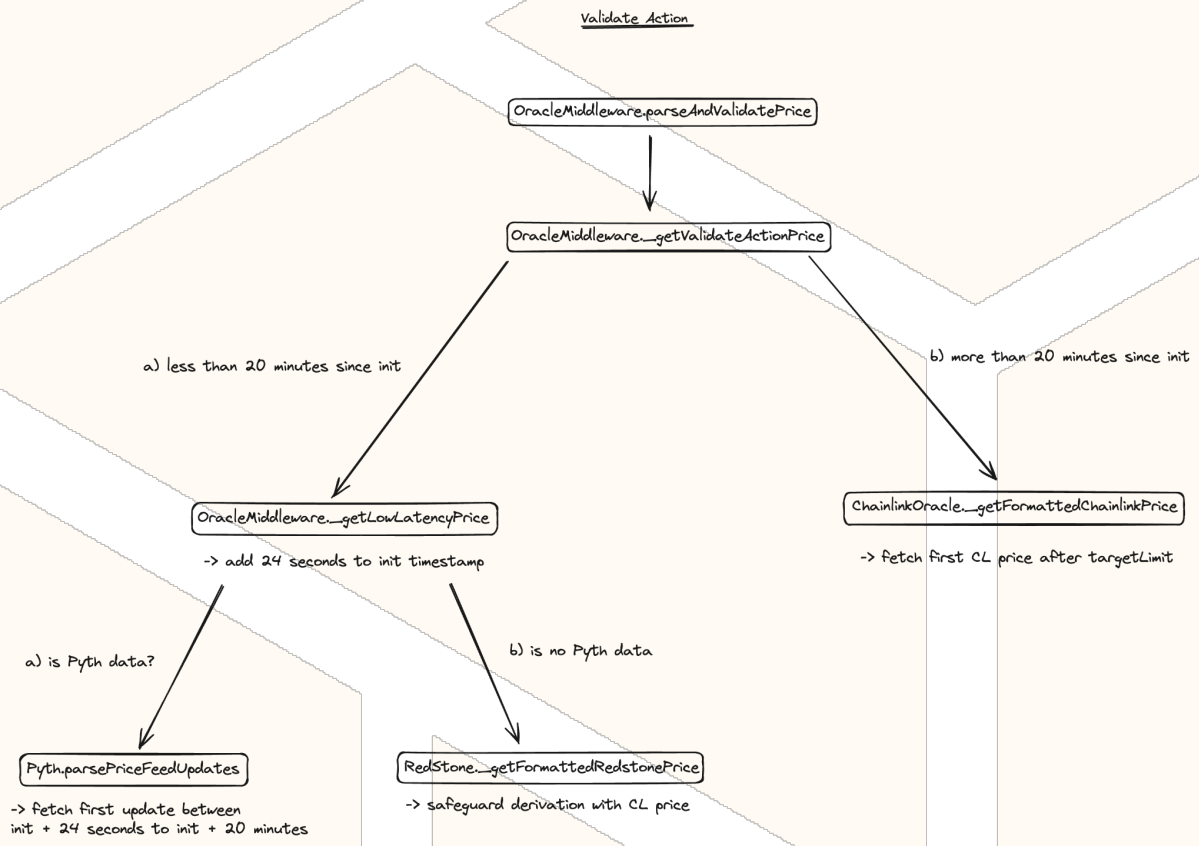
a) **Initiations**: During initiation actions such as deposit, withdrawal, opening and closing interactions, the _getInitiateActionPrice function is invoked. There are two main scenarios:

1. priceData variable was provided: In that scenario, it will fetch the updated Pyth or Redstone price, depending on the provided data.
2. priceData variable was not provided: In that scenario, it will fetch the latest Chainlink and Pyth price and will use the newer of both.

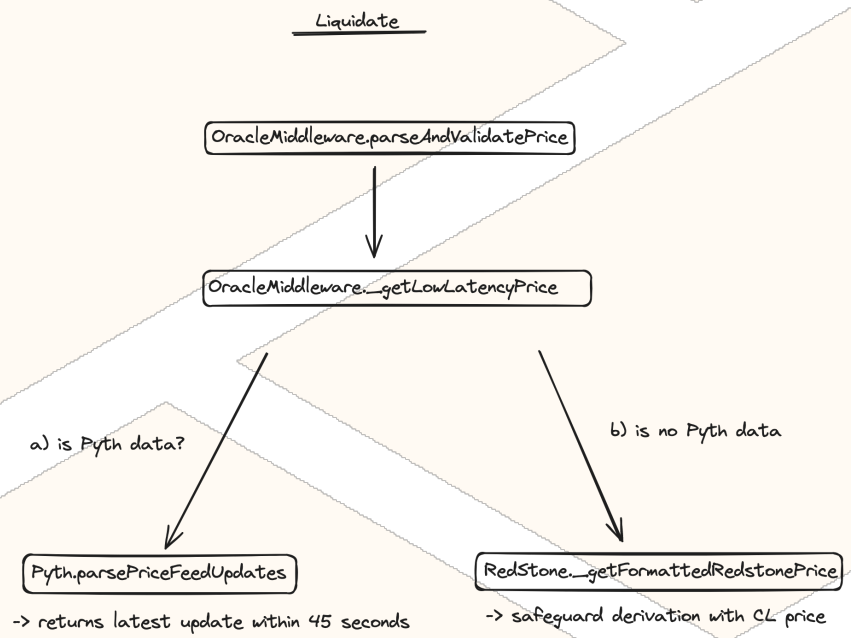


b) **Validations**: During validation actions such as deposit, withdrawal, opening and closing interactions, the _getValidateActionPrice function is invoked. There are two main scenarios:

1. The initiation is less than 20 minutes ago: In that scenario, it will fetch either the Pyth or Redstone price at the initiate timestamp + 24 seconds until up to the initiate timestamp + 20 minutes.
2. The initiation is more than 20 minutes ago: In that scenario, it will fetch the first Chainlink price after the targetLimit



c) **Liquidations:** Whenever a liquidation is happening, it will either consult the Pyth or Redstone oracle with no specific direction.



Privileged Functions

- transferOwnership
- renounceOwnership
- acceptOwnership
- setValidationDelay
- setChainlinkTimeElapsedLimit
- setPythRecentPriceDelay
- setRedstoneRecentPriceDelay
- setConfRatio
- setLowLatencyDelay
- withdrawEther
- setPenaltyBps

Issue_06	Governance Privilege
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, the contract owner can change _penaltyBps up to 1000, which then means a 10% price change.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	

Issue_07	Deduction of validateRoundId will not work if it is the first phaseId for a round
Severity	**High**
Description	Chainlink incorporates specific methodology when it comes to the roundId variable. In short, Chainlink uses different aggregator rounds which are updated occasionally. Each phaseId starts with a new aggregatorRoundId whenever it is updated. A longer summary on this methodology can be found here: https://bailsec.io/tpost/8oi4p03381-about-chainlink-oracle-roundid-phaseid-a If we now take a look at the following code-snippet for the _getValidateActionPrice function: *// previous round id*

```
        ChainlinkPriceInfo memory chainlinkOnChainPrice =
        _getFormattedChainlinkPrice(MIDDLEWARE_DECIMALS,
validateRoundId - 1);

        // if the price is negative or zero, revert
        if (chainlinkOnChainPrice.price <= 0) {
        revert
OracleMiddlewareWrongPrice(chainlinkOnChainPrice.price);
        }

        // if previous price is higher than targetLimit
        if (chainlinkOnChainPrice.timestamp > targetLimit) {
        revert OracleMiddlewareInvalidRoundId();
        }

        // validate round id
        chainlinkOnChainPrice =
_getFormattedChainlinkPrice(MIDDLEWARE_DECIMALS,
validateRoundId);
```

We realize that the previous roundId is just fetched by doing *validateRoundId -1*. This will never work if the validateRoundId is the first round for a fresh phase.

Illustrated:

targetLimit = 524

aggregatorId 1 = 525
aggregatorId 0 = 524

However, aggregatorId 0 can never be fetched because it is from a different round.

	Thus it will revert the whole transaction. This issue has been rated as high severity because it will block validations of actionable positions and thus any interaction, since an actionable position is mandatory to be validated (if applicable).
Recommendations	In that specific scenario, consider to ignore the value of *validateRoundId -1* and just continue with the value of validateRoundId.
Comments / Resolution	

Issue_08	Increased flexibility for users to choose between Redstone and Pyth oracle
Severity	**Low**
Description	For all validation phases, users can choose between the redstone and the pyth oracle (depending on which priceData was provided). This freedom can be arbitraged by users to opt for a better price. Similarly to that it is handled within initiation actions.
Recommendations	Consider if this exposes a risk, if yes, consider sticking with one oracle solution and only use a secondary oracle as fallback oracle.
Comments / Resolution	Resolved, the Redstone implementation will be removed.

Issue_09	Staleness situation of latest CL price is disregarded
Severity	**Low**
Description	The _getLowLatencyPrice function has two different scenarios which are based on the user-provided data parameter: a) Use the Pyth oracle b) Use the Redstone oracle For the latter scenario, the _getFormattedChainlinkLatestPrice function is called. This function is meant to return the chainlink price scaled to 18 decimals and uses this value to cross-reference with the returned RedStone price. However, in the scenario where the price is stale, it will return type(int256).min, which is - 57896044618658097711785492504343953926634992332820282019728792003956564819968 Due to the fact that this value is below 0, the safety check with the Chainlink oracle will be void as it does not enter in the desired condition: *if (chainlinkPrice.price > 0) {* *if (price_.price > uint256(chainlinkPrice.price) * 3) {* *revert OracleMiddlewareRedstoneSafeguard();* *}* *if (price_.price < uint256(chainlinkPrice.price) / 3) {* *revert OracleMiddlewareRedstoneSafeguard();* *}* *}*
Recommendations	Consider reverting if the chainlink oracle returns an invalid price

Comments / Resolution	

Issue_10	Chainlink deviation check within _getLowLatencyPrice is too loose
Severity	**Low**
Description	Currently, within the _getLowLatencyPrice function in the scenario where the Redstone price is fetched, the following deviation check against the CL oracle is being executed: *if (chainlinkPrice.price > 0) {* *if (price_.price > uint256(chainlinkPrice.price) * 3) {* *revert OracleMiddlewareRedstoneSafeguard();* *}* *if (price_.price < uint256(chainlinkPrice.price) / 3) {* *revert OracleMiddlewareRedstoneSafeguard();* *}* *}* This check is certainly too loose and will not be sufficient.
Recommendations	Consider executing a more strict check.
Comments / Resolution	Resolved, the Redstone oracle implementation has been removed.

WstEthOracleMiddleWare

The WstETHOracleMiddleWare contract is an extension of the OracleMiddleWare contract which is responsible to fetch the wstETH price.

Privileged Functions

- transferOwnership
- renounceOwnership
- acceptOwnership
- setValidationDelay
- setChainlinkTimeElapsedLimit
- setPythRecentPriceDelay
- setRedstoneRecentPriceDelay
- setConfRatio
- setLowLatencyDelay
- withdrawEther
- setPenaltyBps

Issue_11	stETH depeg scenario results in higher oracle price
Severity	**Low**
Description	The parseAndValidatePrice function fetches the ETH price and then converts it to wstETH using the stEthPerToken function on the wstETH contract: *// fetched eth price* *PriceInfo memory ethPrice =* *super.parseAndValidatePrice(actionId, targetTimestamp, action, data);* *// stEth ratio for one wstEth* *uint256 stEthPerToken = _wstEth.stEthPerToken();* *// wsteth price* *return PriceInfo({* *price: ethPrice.price * stEthPerToken / 1 ether,* *neutralPrice: ethPrice.neutralPrice * stEthPerToken / 1 ether,* *timestamp: ethPrice.timestamp* *});* Notably, the stEthPerToken function returns how much stETH will be received per wstETH. If now the stETH price depegs, this will not impact this conversion rate but it will result in a flawed return price. **Illustrated:** Scenario 1: - ETH price = 3000e18 - stETH price = 3000e18 - 1 stETH = 1 ETH - stETHPerToken = 1.174 price = 3000e18 * 1.174e18 / 1e18 -> 3522e18

	-> 1 wstETH = 3522e18 USD Scenario 2: - ETH price = 3000e18 - stETH price = 2850e18 - 0.95 stETH = 1 ETH - stETHPerToken = 1.174 price = 3000e18 * 1.174e18 / 1e18 -> 3522e18 -> 1 wstETH = 3522e18 However, in reality 1 wstETH = 3345.9e18 Therefore, in a depeg scenario, the returned wstETH price will be higher than the real wstETH price.
Recommendations	This is considered as a design choice to prevent scenarios where users could manipulate the stETH oracle price due to thin liquidity and exploit the protocol by opening long positions beforehand. Thus the ETH oracle price is used. It is clear that if the stETH price depegs by a large deviation, users should theoretically be liquidated while they are not. However, given the fact that the overall protocol always uses ETH + ratio, it doesn't seem that this would lead to exploitable situations. **One thing has to be noted: The protocol will then assume an incorrect USDN price, which could then open up an arbitrage opportunity with liquidity pools.**
Comments / Resolution	

Oracle

ChainlinkOracle

The ChainlinkOracle contract is a simple oracle handler contract which is inherited by the OracleMiddleware contract and the LiquidationRewardsManager. 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), which is then used throughout the protocol.

For the inheritance with the LiquidationRewardsManager, the following function(s) will be used:

- getChainlinkLatestPrice

Privileged Functions

- none

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

Privileged Functions

- none

Issue_12	Users may receive different price from targetTimestamp price
Severity	**Low**
Description	Whenever a validation occurs, the _getPythPrice function is invoked (if a valid priceData was provided). This will then enter the following else clause: *else {* *// we want to validate that the price is exactly at `targetTimestamp` (first in the second) or the next* *// available price in the future, as identified by the prevPublishTime being strictly less than* *// targetTimestamp* *// we add a sanity check that this price update cannot be too late (more than `_lowLatencyDelay` seconds* *// late) compared to the desired targetTimestamp* *priceFeeds = _pyth.parsePriceFeedUpdatesUnique{ value: pythFee }(* *pricesUpdateData, feedIds, uint64(targetTimestamp), uint64(targetLimit)* *);* *}* The parsePriceFeedUpdatesUnique function then fetches the first price between targetTimestamp and targetLimit (targetTimestamp + 20 minutes). In such a scenario where no price update has occurred for a longer time, the price at the targetLimit or near the targetLimit can be returned. This means the original idea of using the price at the targetTimestamp is not followed.
Recommendations	Since this requires a certain inactivity level on the Pyth side we do not consider it as a direct exploitable risk. However, it should definitely be kept in mind that such a scenario (to advantage of a malicious user)

	can occur.
Comments / Resolution	

Issue_13	Unsafe uint64 casting may result in overflow if pythPrice is > uint64
Severity	**Informational**
Description	Within the _formatPythPrice function, the pyth price is downcasted as follows: *uint256(uint64(pythPrice.price))* unsafe casting to uint64 will never work if the pyth oracle returns the price with 18 decimals instead of 8 decimals, as the result will then silently overflow. **However, due to the fact that the pyth price itself is from type int64, this operation can be considered as safe:** https://github.com/pyth-network/pyth-sdk-solidity/blob/main/PythStructs.sol#L15
Recommendations	Consider keeping this in mind if the pyth oracle implementation changes in the future.
Comments / Resolution	

RedstoneOracle

The RedstoneOracle contract is responsible for fetching the price from the corresponding Redstone oracle by carefully extracting the provided calldata. It furthermore normalizes the return value to 18 decimals with the inherent assumption that the return value is scaled by 8 decimals.

Whenever the Redstone oracle is consulted to fetch a price for an action initiation, the last Redstone update must not be older than 10 seconds (_redstoneRecentPriceDelay).

Whenever the Redstone oracle is consulted to fetch a price for an action validation, the last Redstone update must be “in a 1-heartbeat window starting at the target timestamp”, whereas the heartbeat is hard coded as 10 seconds.

Disclaimer: The Redstone Oracle architecture is out of scope, therefore this section has three core assumptions:

a) _extractPriceUpdateTimestamp extracts the correct timestamp
b) getOracleNumericValueFromTxMsg extracts the correct value for the corresponding feedId
c) Users can not craft malicious calldata to manipulate the oracle return value, such as receiving an outdated or simply incorrect price

We highly recommend seeking coverage, specifically for the calldata extraction process.

Privileged Functions

- none

Issue_14	Insufficient Redstone documentation
Severity	**Medium**
Description	This issue is not related to the Smardex implementation but rather relates to the fact that Redstone itself does not provide accurate data which describes the calldata extraction process. Without fully auditing Vendored, it is impossible to guarantee the correctness of the integration.
Recommendations	Consider either removing the Redstone oracle (in our opinion CL and Pyth are sufficient) or consider seeking full coverage over the Vendored scope, which ensures in the disclaimer explained assumptions.
Comments / Resolution	

Rebalancer

Rebalancer

The Rebalancer contract is a helper contract to support the core with rebalancing the protocol if the vault side becomes forcefully dominant due to liquidations. In that scenario, the rebalancer will open a long position with a specific leverage that meets the desired target trading exposure, while ensuring that all funds are used which are deposited in the rebalancer.

Since the rebalancer is now considered as a participant on the long side, funds are needed to be able to fill that position. Therefore, users can deposit wstETH to the rebalancer which is then used to create a long position.

The following interactions can be executed by users:

a) Deposit wstETH which then is considered for the next long position creation (trigger), using an initiate -> validate approach.

b) WIthdraw wstETH which was just deposited for initiation but not yet validated.

c) Withdraw wstETH which was already validated but not yet processed, using an initiate -> validate approach.

c) Fully or partially withdraw wstETH from an existing position

Similar to the core, an initiate and validate structure was implemented which counters any potential frontrunning scenarios. Each interaction must be initiated first and can then be validated 24 seconds afterwards.

Users will gain PnL based on the wstETH price of position opening and closing as well as additional incentives as share of the liquidation penalty.

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:
-> 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.

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.

Privileged Functions

- transferOwnership
- renounceOwnership
- setPositionMaxLeverage
- setMinAssetDeposit
- setCloseImbalanceLimitBps
- setTimeLimits

Issue_15	Blunder within initiateClosePosition allows users to withdraw their liquidated positions
Severity	**High**
Description	Whenever the rebalancer position is considered as liquidated during the _triggerRebalancer function within the ProtocolLong library, the returned position value during the _flashClosePosition value will be zero: *if (posId.tickVersion != version) {* *return positionValue_;* *}* This means that the position from the rebalancer and all deposited assets thus far (not assets counted in _pendingAssetsAmount) are basically lost, users should not get any funds back therefore. If now the updatePosition function in the rebalancer is called, the previousPosValue parameter will be zero, while the opening amount was non-zero, which means the function enters the else block in the following code: *if (previousPositionData.amount > 0) {* *// if the position has not been liquidated* *if (previousPosValue > 0) {* *// update the multiplier accumulator* *accMultiplier = FixedPointMathLib.fullMulDiv(* *previousPosValue,* *previousPositionData.entryAccMultiplier, previousPositionData.amount* *);* *} else {* *// update the last liquidated version tracker* *_lastLiquidatedVersion = positionVersion;*

	} } Furthermore, the new position data is simply determined by the default accumulator and the _pendingAssetsAmount. Due to a blunder within the initiateClosePosition function, it now erroneously allows users to withdraw their position amount even if the position was liquidated, which means depending on the newly deposited amount as well as future deposits, users can now steal funds from other users.
Recommendations	Consider checking if a position is deemed as liquidated and revert in that scenario.
Comments / Resolution	

Issue_16	Lack of grace period allows users to profit from rebalancer bonus risk-free
Severity	**High**
Description	Whenever the rebalancer is triggered, all assets that have been previously deposited by users are incorporated in the new position opening and commingled with the previous position. On top of this comes the rebalancer bonus which then forms the final position value for the rebalancer position within the USDN protocol. Users that have just deposited can then immediately withdraw their deposit, including a part of the bonus which is calculated pro-rata: > amountToCloseWithoutBonus + amountToCloseWithoutBonus * (amountLongInUSDN - amountInRebalancer) / amountInRebalancer While, theoretically, that bonus is meant to be allocated only to these users that have already had processed deposits. Therefore, users can effectively steal a part of the bonus, nearly risk-free.
Recommendations	Consider incorporating a minimum deposit time. That could mean users can withdraw only after 3 iterations.
Comments / Resolution	

Issue_17	Unused variables
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers. *uint256 internal _closeImbalanceLimitBps = 500;*
Recommendations	Consider removing the unused variables
Comments / Resolution	

USDN

Usdn

The USDN contract is a modified ERC20Permit rebase token with an ever increasing supply and burnable functionality. The main goal of this token is to stay between an USD value of 1.0084 and 1.009 and whenever the threshold of 1.009 is surpassed, a rebase happens which increases the supply to match the target price of 1.0084. The calculation to determine the new supply is as follows:

targetSupply = vaultBalance * (wstETHPrice) / (1.0084e18)

which is mathematically derived from the price calculation:

price = vaultBalance * wstETHPrice / usdnSupply

A rebase is potentially triggered upon all important interactions such as:

a) Deposit initiation / validation
b) Withdraw initiation / validation
c) Opening initiation / validation
d) Closing initiation / validation
e) Liquidation

While this token still keeps the full ERC20 standard including:

- burn
- burnFrom
- transfer
- transferFrom

It exposes additional logic which allows the handling of shares via:

- burnShares
- burnSharesFrom

- transferShares
- transferSharesFrom

The token storage is completely denominated in shares which means that the actual token supply is only relevant for external protocols.

Privileged Functions

- grantRole
- revokeRole
- mint
- mintShares
- rebase
- setRebaseHandler

Issue_18	The rounding to the closest value is erroneous in some cases
Severity	**Low**
Description	In some rare edge cases, the rounding to closest rounds up instead of rounding down. For example: amount = 1500000001 d = 1000000001 tokensDown = amount / d = floor(1.4999999995) = 1 remainder = amount % d = 500000000 tokensUp = 1 + 1 = 2 half = d >> 1 = 500000000 -> remainder >= half -> return tokensUp = 2 This returns 2 instead of 1.

	The main culprit is that when d is divided by 2, it rounds down.
Recommendations	Consider rounding half up to make sure that the rounding to closest is always accurate.
Comments / Resolution	

Issue_19	Sudden rebase may result in DoS when using transferFrom with non-unlimited approvals
Severity	**Low**
Description	Whenever a rebase is triggered, this will increase the totalSupply while decreasing the price. Simplified, if a user has now 100 tokens worth 100 USD, a rebalance will result in the user having now 200 tokens worth 100 USD. The token price has thus decreased and the balance increased but the user still holds the same nominal USD value. This will become a problem with the transferFrom usage paired with non-unlimited approvals. If a contract attempts to spend 100 USD of tokens while the user has granted an approval for 100 tokens and now the rebase has happened, the contract attempts to spend 200 tokens (since these are worth 100 USD). The transferFrom call will now revert because the user has only approved 100 tokens. This can specifically become a problem if a user wants to withdraw from the vault shortly after a rebase has happened because the allowance will now attempt to spend 200 tokens instead of 100 tokens for the same amount of shares.
Recommendations	This is an architectural issue which stems from the fact that shares are used to determine a user's balance while the allowance is being denominated via tokens.

	We do not recommend a change but rather spread awareness to the community for such possible scenarios.
Comments / Resolution	

Issue_20	balanceOf rounding is inconsistent
Severity	**Informational**
Description	The balanceOf function will be used in multiple different DeFi protocols, which means that the correctness of this function is very sensitive. In the current state the rounding is inconsistent and simply rounds to the closest value - either up or down. While this inconsistency does not have a major impact because we are talking about a negligible value of 1 wei, it may still happen that this will have unforeseen impacts whenever USDN is used in other protocols. For example if depositing into other contracts, it may be possible that the contract afterwards displays 1 wei less than anticipated (if there is an existing balance beforehand which then results in rounding down once the new transfer has happened)
Recommendations	Consider staying consistent in the rounding direction and rounding down.
Comments / Resolution	

Issue_21	_rebaseHandler should execute sync logic
Severity	**Informational**
Description	This is not an issue in itself but rather a reminder / recommendation to execute sync logic on dominant UniswapV2 pairs in an effort to not allow exploiters to steal the rebased USDN amount from the pair.
Recommendations	Consider invoking sync after the divisor has been reduced.
Comments / Resolution	

Issue_22	Inconsistency in share overflow safeguard between _update and _updateShares
Severity	**Informational**
Description	The _updateShares function simply increases _totalShares as follows: *_totalShares += value* Whereas the _update function handles this logic as follows: *unchecked {* *uint256 res = _totalShares + valueShares;* *// overflow check required, the rest of the code assumes that `totalShares` never overflows* *if (res < _totalShares) {* *revert UsdnTotalSupplyOverflow();* *}* *_totalShares = res;* *}*

	In the first example, the overflow is guarded due to solidity's built-in overflow protection, whereas in the second example, solidity's built-in protection is void due to the unchecked clause. This redundantly increases complexity while it could just simply follow the first example.
Recommendations	Consider removing the unchecked clause and follow the same mechanism as in the first example.
Comments / Resolution	

Wusdn

The Wusdn contract is a simple wrapper contract that allows users to wrap their USDN and receive the Wusdn token as receipt. Users can either wrap via directly providing their desired share amount and invoke wrapShares or via providing their desired token amount via the wrap function.

Whenever desired, users can then unwrap their Wusdn and receive the exact same share amount as initially provided. The rationale behind this contract is to provide a stable alternative to the USDN token which does not increase in balance. This token can be safely used among DeFi protocols without the disadvantage that comes with rebasing tokens.

Whenever shares are wrapped, users will receive their provided share amount divided by 1e18 and whenever Wusdn is unwrapped, users will receive their provided share amount multiplied by 1e18.

This will always follow this formula:

WUSDN = USDNShares / 1e18

USDN = WUSDN * 1e18

The _wrapShares function has an implemented safeguard that truncates the provided share amount to ensure users will not lose any tokens due to rounding.

Privileged Functions

- transferOwnership
- renounceOwnership

Issue_23	Users may have leftover USDN shares after wrapping shares
Severity	**Informational**
Description	The _wrapShares function calculates the corresponding WUSDN amount based on the provided USDN share amount as follows: *wrappedAmount_ = usdnShares / SHARES_RATIO;* Afterwards, the wrappedAmount is then multiplied by 1e18 to receive the actual USDN share amount which is being transferred in. This is a safeguard to prevent users from accidentally donating dust to the contract and ensures that the provided USDN share amount is always a multiple of 1e18. If for example a user provides 1.5e18 USDN shares, he will actually only transfer 1e18 in and receive 1 WUSDN. This safeguard will have the side-effect that not all provided shares will be consumed.
Recommendations	We do not recommend a change. This is solely to keep the report complete and point out all important facts/side effects.
Comments / Resolution	

USDN Protocol

The USDN Protocol is the core path of this audit report. The codebase was written with a modular approach which outsources most of the logic to the library contracts. These libraries are then called by the Core Contracts and will handle the business logic.
The core protocol mainly involves two parts:

a) Vault: This part of the protocol allows users to mint USDN by providing wStETH and vice-versa.

b) Long: This part of the protocol allows users to long ETH by providing wStETH

The main idea behind the protocol conception relies on the correctness of the funding rate, as users will be incentivized to long ETH whenever the participation in the Vault part of the protocol is high.

Moreover, the protocol exposes various safety and balance mechanisms which attempt to keep the USDN price stable. **However, depending on market circumstances, these mechanisms may not be sufficient.**

Library Contracts

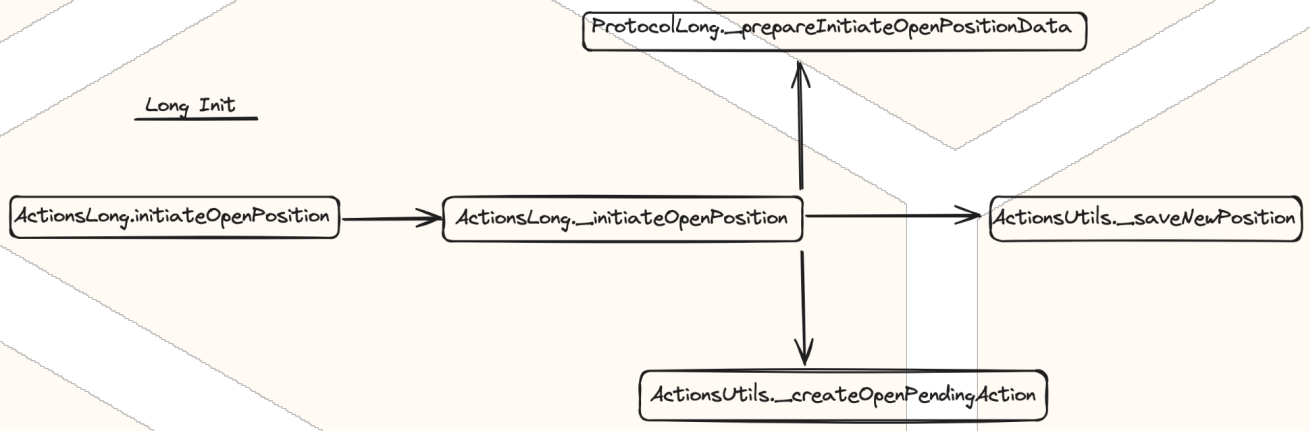
The Library Contracts module incorporates all core functionalities for the USDN protocol and is used as a logic module by the Core Contracts.

ActionsLong

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.

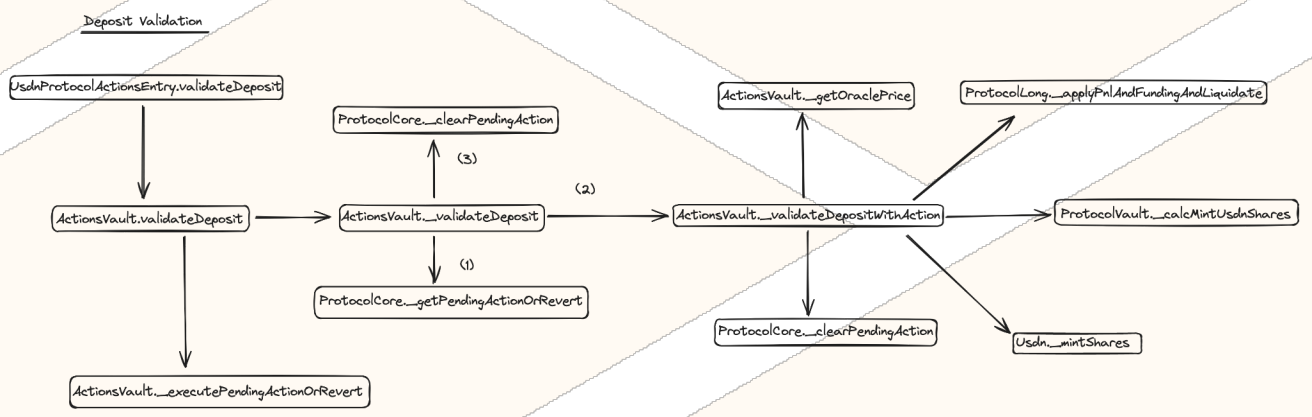
Notably, the following functionality is exposed:

a) **Opening Initiation:** This is the first function which is invoked upon the position creation by the UsdnProtocolActionsEntry contract. 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.



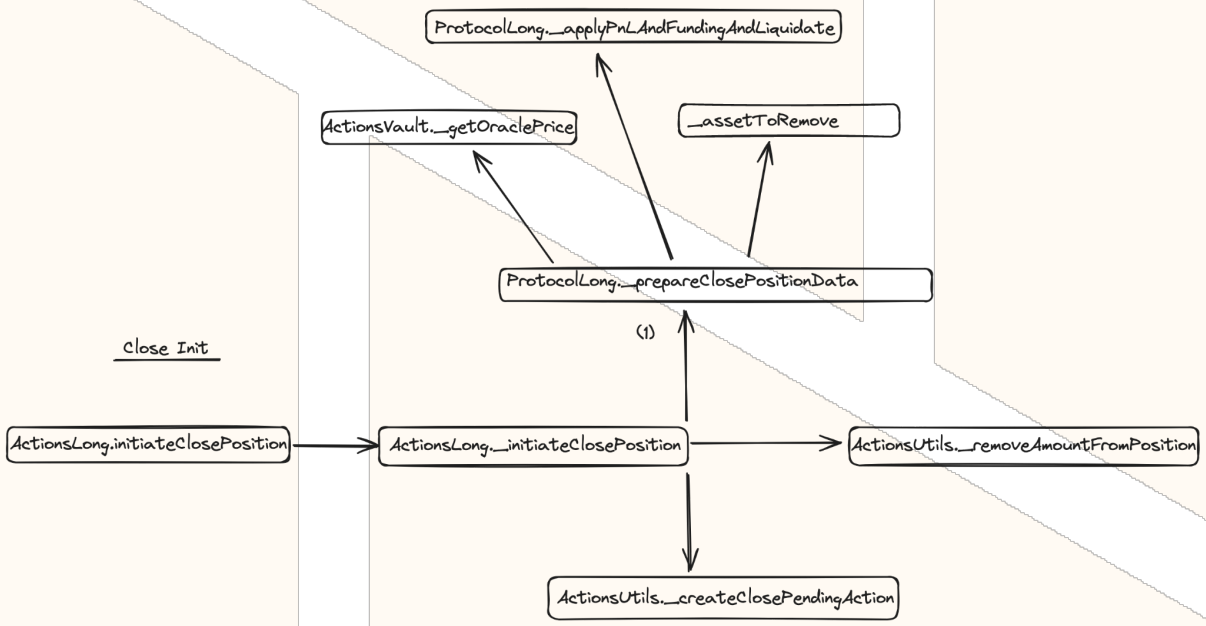
b) **Opening Validation:** Whenever a position opening has been initiated, it is sitting within the _pendingActionsQueue, waiting to be validated by the validator or other users at shortest 24 seconds after the initiation timestamp. The validation will simply use the new price, re-adjust the position expo and mark the position as validated, finally.

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”.



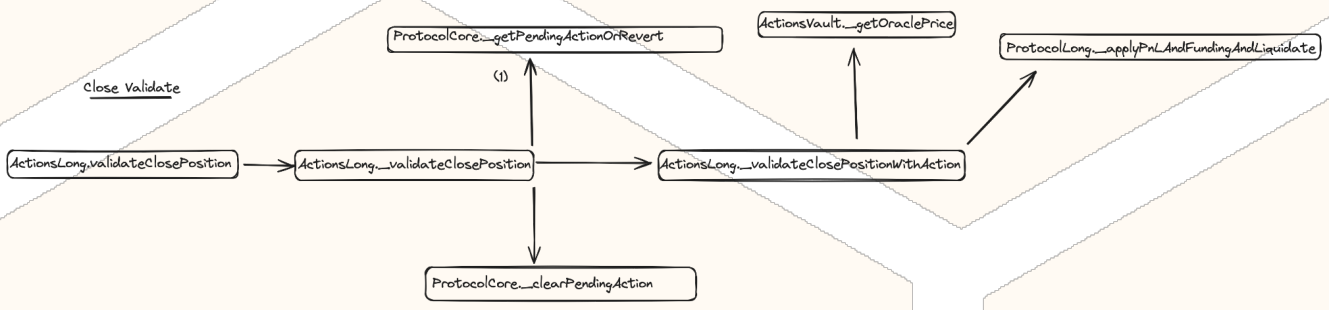
Closure Initiation: Whenever a position opening has been fully validated, the position can then be again fully or partially closed. The closure initiation already fully removes a position from the system or decreases it, which includes updating all corresponding global data plus the tick where the position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

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. Adjustements/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 totally 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.

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

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**, 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.**

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.

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

More elaboration on the math can be found in this link:

https://github.com/SmarDex-Ecosystem/usdn-contracts/pull/206

Appendix: MaxLeverage Excess

Whenever a position has been initiated, it can be validated by other users or the validator. There are two points to consider between the initiation and the validation:

a) The price fluctuates
b) Funding is applied on the protocol

Both a&b can result in the leverage of the position to exceed maxLeverage. In such a scenario, the tick of a position will be adjusted to decrease the leverage back to maxLeverage, using the following steps:

a) Calculate nominal liquidation price without penalty, based on the current price and the maxLeverage:

   (startPrice - (10^21 * startPrice /100000000000000000000))

b) Unadjust the price to reflect the price where the position should be stored and convert it to the corresponding tick:

   desiredLiqPrice * _liqMultiplierAccumulator / (assetPrice*longTradingExpo)

c) Add the penalty on top of the unadjusted price/tick to get the final tick where the position is stored:

   tickToStore = tickWithoutPenalty + currentLiqPenalty + _tickSpacing

Since it may be possible that the tick where the position will now be stored has a different penalty than the current global penalty, a small trick was implemented which calculates the liqPriceWithoutPenalty by subtracting the penalty where the position is going to be stored to calculate the exposure.

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).

Privileged Functions

- none

Issue_24	Malicious user can drain the protocol by using the validator withholding technique and creating risk-free trades
Severity	**High**
Description	In another issue we have raised the fact that validators can simply disable the fallback function which then grants them exclusivity to validate init actions within the first 90 minutes. While this seems like a design choice, as per Smardex team, it allows a malicious user/validator to steal from the protocol by abusing a price arbitrage from the oracle and creating risk-free future trades. Whenever a position is validated, the price from the init timestamp + 24 seconds is meant to be used to open the position. Now throughout the report, we have already elaborated multiple times that (if an initiated action was not validated within 20 minutes), the Chainlink price at init timestamp + 20 minutes is used to validate a position. (For reference see graphic within OracleMiddleware section) This can be exploited as follows: a) Malicious user opens a long position at TS = 100000 b) Malicious user waits 19 minutes until TS = 101140 and assess the current price compared to the price at 100024 (actionTimestamp + _validationDelay) c) If the price at TS = 101140 is larger than price at TS = 100024, immediately validate the initiate action using the price at TS = 100024, otherwise wait 1 minute and validate the position with price at TS = 101140 (to prevent potential loss). d) The user successfully profited from a risk free trade **Illustrated:** **Status Quo:**

position1.tick = 79214
position1.totalExpo = 10e18
position2.tick = 79214
position2.totalExpo = 10e18
_balanceVault = 10e18
_balanceLong = 2e18
_totalExpo = 20e18
currentPrice = 3000e18
_liqMultiplierAccumulator = 5.40e40
_tradingExpo = 18e18

1. Alice initiates a new position with the following parameters:

> validator = maliciousContract
> amount = 1e18 (ignore imbalance, this is to showcase impact)
> desiredLiqPrice = 2754e18

this will result in the following storage:

position1.tick = 79214
position1.totalExpo = 10e18
position2.tick = 79214
position2.totalExpo = 10e18
pos3.tick = 79214
pos3.totalExpo = 10e18
_balanceVault = 10e18
_balanceLong = 3e18
_totalExpo = 30e18
currentPrice = 3000e18
_liqMultiplierAccumulator = 8.10e40
_tradingExpo = 27e18

2. 19 minutes have surpassed and the price increases to 3300e18, due to the validator withdrawal technique, this position cannot be validated by anyone besides the validator

	3. The validator now validates the position with an accurate priceData parameter to fetch the price at init timestamp + 24s > eventual funding could have been applied in the meantime, this can be negligible or even positive for the position, we ignore it for this PoC, same counts for PnL but that does not manipulate our adjusted liquidation price (which is used to calculate the position value) > it may even possible that no funding/PnL has been applied in the meantime (if no user interacted with the protocol) > startPrice = 3000 (ignore fee for simplicity) > expoBefore = 20e18 > expoAfter = 20e18 > no changes are made to the position 4. Position has been created with price = 3000 but real price = 3300 > calculate position value > position.totalExpo * (currentPrice - liquidationPriceNoPenaltyAdjusted) / currentPrice > 20e18 * (3300e18 - 2700e18) / 3300e18 > profit = 2.636 ETH Combined with a similar exploit on position closure, this offers up to 40 minutes of risk-free trades. **While we have added a separate issue on the position closure impact, there are also several other scenarios where this could be exploited, which we will not additionally raise because the Smardex team already indicated that this root-cause will be fixed.**
Recommendations	There are multiple different solutions to fix this issue. The most trivial solution would be to simply wrap the ETH into WETH if the call reverts.
Comments / Resolution	

Issue_25	Edge-Case due to maxLeverage excess allows users to manipulate contract state and benefit with own vault deposit without affecting initiated position with amplification due to validator withholding
Severity	**High**
Description	Whenever a position has been initiated, it was initiated in such a manner that it already serves as a full long position, it will manipulate the funding rate but at the same time it will also be influenced by the funding rate. So far, everything in this logic is perfectly fine, as validating this position will not change the initially stored tick, thus guaranteeing that the position already experiences the funding rate application. If that would not be the case, users could exploit the fact that the position opening already alters the funding rate but once validated, the opened position would be “reset”, thus not experiencing the funding rate since initiation. After some analysis, we in fact identified such a state where the position is “reset”, as in stored at a new tick. This is exactly the case when the maxLeverage is surpassed. There are now three scenarios when this can happen and all have the ground assumption that the initial leverage was already near 10: a) The startPrice has decreased, this will increase the leverage b) Funding has been positive, this will increase the leverage c) A mix of a&b From a/b/c, it is now already possible that we can “estimate” the future funding rate and thus we can take a directional bet on this scenario.

	This now allows a user the following: a) Create a vault position which benefits from positive funding b) Create a long position with near 10x leverage, this action assumes that the maxLeverage will be exceeded during the validation c) During the validation, maxLeverage is now exceeded, it will automatically calculate the new “real/adjusted” liquidation price based on the startPrice and then unadjust this price to store the position at the corresponding tick. Now it is important to understand that this unadjustment will be done using the updated contract state, hence this position will be stored as “newly created”. The exploiter now successfully created a large long position which impacts the funding rate in an effort to increase his own vault position while after the validation, the created long position does not negatively suffer from the increased funding rate. This issue alone is not yet sufficient because this means that the user could benefit 24 seconds long (which alone is already a large problem). Additionally to that, the validator can withhold any validation due to disabling the fallback logic and thus not receiving ETH. After 20 minutes then, the validator can enable ETH transfers and validate the action which means that the Chainlink oracle will be considered which uses the timestamp of init + 20 minutes, thus manipulating the contract 20 minutes long.
Recommendations	Consider removing the maxLeverage scenario for position validations as there is not even a disadvantage for users with a leverage that is larger than 10x.
Comments / Resolution	

Issue_26	Edge-case of initiated and liquidated position opening in combination with lowered _securityDeposit allows malicious user to steal security deposit from validator before the 90 minutes threshold
Severity	**High**
Description	Throughout the protocol, there are two main scenarios for validating an initiated action: a) Validating an action within the first 90 minutes via the standard validate functions, such as: - ActionsLong.validateOpenPosition - ActionsLong.validateClosePosition - ActionsVault.validateDeposit - ActionsVault.validateWithdrawal Whenever these functions are invoked, the security deposit will always be transferred to the validator address. Notably, this can be done even after 90 minutes. b) Validating an action after 90 minutes via: - ActionsVault._executePendingActionOrRevert (mandatory after each initiation/validation) - ActionsUtils.validateActionablePendingActions Whenever these functions are invoked, the security deposit goes straight to the caller, contrary as in a), where it goes to the validator. Therefore, the desired idea is clear: Within the first 90 minutes, the security deposit should go towards the validator, after 90 minutes, the security deposit should go to the caller. This is however not the case if an initiated position was already liquidated and a malicious user adds a new initiated position to this validator. Even if there are less than 90 minutes surpassed, the refund

	is happening to the caller: https://github.com/SmarDex-Ecosystem/usdn-contracts/blob/ff2b96fefc6ab41bb2adccef430c1f2f19e29dda/src/UsdnProtocol/libraries/UsdnProtocolCoreLibrary.sol#L656 It is now clear that the newly initiated action will be added to the validator from whom we just "stole" the security deposit and therefore this validator can just take "our security deposit". However, in the scenario where the _securityDepositValue variable has been lowered within the ProtocolSetters library, it opens the attack path for a malicious user to now benefit from this deviation.
Recommendations	Consider refunding the securityDeposit to the validator within the first 90 minutes under all circumstances. This will require some slight refactoring of the logic.
Comments / Resolution	

Issue_27	Opening Initiation without priceData can result in immediate gain for users
Severity	**High**
Description	The general rule for any state transition within the architecture is that the contract state should always be up to date. This is specifically important for PnL and funding applications. If users initiate a position opening, they can provide a priceData parameter, which is however not mandatory to provide. To illustrate this issue, we can consider a simple opening initiation and its corresponding oracle fetching mechanism. Upon this action, it will always invoke the _getInitiateActionPrice function. The _getInitiateActionPrice function has several different outcomes, whereas for most scenarios, not the exact current timestamp is returned. In the scenario of a valid Pyth price update, it may actually be possible that the current block.timestamp is returned. For all other scenarios, (potentially) not the most recent price is used but a price which is a bit delayed. This depends on the configured allowed delays. In the scenario of such a delay, the returned timestamp will not be the current timestamp. This is now specifically important in how the returned timestamp will be used: *(, data_.isLiquidationPending) = Long._applyPnlAndFundingAndLiquidate(* *s,* *currentPrice.neutralPrice,* *currentPrice.timestamp,*

	s._liquidationIteration, *false,* *ProtocolAction.InitiateDeposit,* *currentPriceData* *);* The marked timestamp is then used to calculate the funding. This means in the worst case scenario, users can theoretically calculate the funding rate (which is to be applied until the most recent timestamp), initiate a position opening while only updating the funding rate with the returned timestamp and then later potentially profit from the funding rate which will be applied **after** the position has been created (the position will be created with the init state, even if validated much later). This means that users can exploit the potential discrepancy between the returned timestamp and the actual timestamp and its impact on the funding rate. *This issue can be 1:1 applied on the deposits Additionally but less impactful is the position closure scenario where users can actually avoid the funding application up to the recent timestamp.
Recommendations	Consider making it mandatory for users to initiate a deposit with the most recent block.timestamp as the latest price update.
Comments / Resolution	

Issue_28	Malicious users can drain funds from the protocol via an already closed position
Severity	**High**
Description	In a similar fashion as described within *"Malicious user can drain the protocol by using the validator withholding technique and creating risk-free trades"*, it is possible to simply initiate a position closure with a malicious validator. This validator can then withhold the validation up to 19 minutes and determine the price at this time. If the price is favorable (it has increased), the validator can simply wait 1 more minute and validate it via the Chainlink price. If the price has decreased, the validator can immediately validate the closure using the Redstone/Pyth price at init time + 24 seconds. In the first scenario where the price has increased, this will result in a recalculation of the received position amount: > positionTotalExpo * (currentPrice - liqPriceWithoutPenalty) / currentPrice In numbers: > A user has 10x leverage and has deposited 1e18 wstETH into the contract > The position is closed without a profit/loss; current price = 3000 > 19 minutes later, current price is 3300e18, user waits until 20th minute and validates the closure, new position value is calculated:

	> User now receives 1.818 ETH instead of 1 ETH Combined with a similar exploit on position openings, this offers up to 40 minutes of risk-free trades.
Recommendations	Consider transferring out WETH instead of ETH to the validator in the scenario when the call reverts. This will prevent the withholding technique.
Comments / Resolution	

Issue_29	Edge-case within _validateClosePositionWithAction will result in incorrect position value if penalty is changed after closure initiation
Severity	**High**
Description	The _validateClosePositionWithAction function is responsible for validating a position closure with the main goal to deduct or increase the amount of assets which will be transferred to the recipient for the closure. The position value is calculated following the standard known formula: > position.totalExpo * (currentPrice - liquidationPriceNoPenaltyAdjusted) / currentPrice Whereas liquidationPriceNoPenaltyUnadjusted is calculated in the following line: *int24 tick = Long._calcTickWithoutPenalty(s, long.tick, Long.**getTickLiquidationPenalty**(s, long.tick));* For importance is the **getTickLiquidationPenalty** function as this either returns the default penalty or the penalty which is already assigned to the tick. In the edge-case where: a) The initiated position was the only position in the tick (which was removed by now) b) The default penalty is changed after the initiation , the penalty which is returned by this function will be different to the real penalty which was previously allocated, which then in turn results in an incorrect liquidationPriceNoPenaltyAdjusted value and thus an incorrect position value.

Recommendations	Consider caching the penalty which was previously used.
Comments / Resolution	

Issue_30	Position closure without latest funding incorporation can be used to escape mandatory liquidation
Severity	**Medium**
Description	Whenever a position is closed, this is done via the initiateClosePosition function. We have already elaborated multiple times that this allows for some oracle arbitrage in the scenario where not the most recent block.timestamp is returned by the oracle. In this scenario, the funding will only be updated up to the potentially outdated timestamp, thus not incorporating the latest funding application. Due to the fact that during the validation, the same state as during the initiation is used (closeLiqMultiplier), a user can now successfully close a position without incorporating this latest funding period, which is advantageous if the funding was positive. This can furthermore be used to potentially “escape liquidations” which should theoretically have happened (due to funding incorporation).
Recommendations	Consider not allowing for oracle arbitrage but forcing users to always update and fetch the newest Pyth price.
Comments / Resolution	

Issue_31	Lack of deadline parameter can result in stuck position in mempool and undesired position value/leverage altering
Severity	**Medium**
Description	Whenever a position opening is initiated, this position is stored at a very specific tick. Once a position has been initiated, it is actually already a fully opened position, which means it can technically be liquidated. If users now invoke the initateOpenPosition function with the desired parameters, this transaction will remain in the mempool until it has been picked up. In the meantime, it is possible that the price shifts downwards to near the desired liquidation price, this will not only unexpectedly increase the desired leverage but also bring the position near liquidation and decrease the position value. Of course, vice-versa it will also result in decreasing the desired leverage.
Recommendations	Consider incorporating a deadline parameter to automatically revert the transaction if it remains stuck for too long in the mempool.
Comments / Resolution	

Issue_32	Lack of exclusivity for validator validation can result in worse outcome
Severity	**Medium**
Description	As we have already outlined within the OracleMiddleware section, there are two possibilities to fetch the price within the first 20 minutes after the initiation, which is either the Redstone case or the Pyth case. Now we need to understand that, while the validator will receive the 0.5 wStETH within the first 20 minutes, anyone can validate an initiated action. This means also that anyone can provide the priceData parameter which is responsible for fetching the price and timestamp. A malicious user can therefore inspect which oracle has the least favorable price for the initiated action and can then use a carefully crafted priceData to fetch this price, which then results in an opportunity loss for the initial caller.
Recommendations	Consider either not allowing users to validate any initiated action or only allow to consider one price oracle.
Comments / Resolution	

Issue_33	Long positions are inherently +EV due to stETH rebases
Severity	**Medium**
Description	Due to the fact that the stETH balance is ever-increasing and thus wstETH increases yearly by roughly 4% (at the time of writing), creating a leveraged long position will be always +EV even if the market does not move. If a user deposits 100 wstETH with 10x leverage, he will have a nominal position value of 134 wstETH after 1 year (if the price of ETH has not moved). At the same time, this means that vault depositors will bear this loss.
Recommendations	This seems to be a design choice with its root-cause of using the wstETH price as oracle price. Fixing this issue would mean that the ETH price must be used to determine any PnL, however, that would then not properly reflect the perpetual contract of wstETH. It is possible that the funding fee regulates this solely without any further needed intervention.
Comments / Resolution	

Issue_34	Sudden price spikes are not incorporated into imbalance checks opening validation
Severity	**Low**
Description	First of all, we need to explain how new positions are added and how the imbalance checks come into play with this. Whenever a new position is added, this means the initiateOpenPosition function is invoked. During this flow, all important variables are crafted and states are updated. Most importantly for this example, the position exposure is calculated: *data_.positionTotalExpo = _calcPositionTotalExpo(amount, data_.adjustedPrice, liqPriceWithoutPenalty);* *totalExpo_ = FixedPointMathLib.fullMulDiv(amount, startPrice, startPrice - liquidationPrice).toUint128();* Afterwards, the imbalance calculation is done as follows: *(longTradingExpo - vaultBalance) * 10_000 / vaultBalance* Now we need to explain each variable: longTradingExpo: The “borrowed” wstETH amount vaultBalance: Nominal vault balance The problem: During the validation phase, the exposure will change due to the starting price change, remember the following formula for the expo calculation: amount * startPrice / (startPrice - liqPriceNoPenalty)

	this is then applied on the _totalExpo and will therefore change the tradingExpo which can result in exceeding long dominance upon position opening: *s._totalExpo = s._totalExpo + expoAfter - expoBefore;* However, there is no further imbalance check which means that the protocol can run into imbalances during large price spikes/drops.
Recommendations	Consider executing an additional imbalance check during the validation phase. However, this is a non-trivial task as logic must be implemented that properly refunds the position in a scenario where the imbalance limit is reached in order to avoid stuck positions. Such a mechanism must be carefully audited.
Comments / Resolution	

Issue_35	Redundant usage of closeLiqMultiplier
Severity	**Low**
Description	Within the *Appendix: Fixed Precision Multiplier*, we have explained the idea behind the closeLiqMultiplier. It goes without saying that this logic adds a lot of complexity to the system which can ideally be avoided by simply storing the adjusted tick (with or without) penalty into the LongPendingAction struct during the closure initiation.
Recommendations	At Bailsec, we are of the opinion that with increased code complexity the risk of vulnerabilities increases exponentially. Therefore, it often makes sense to reconsider certain design choices and eventually replace them with a more simplified approach. Consider simply storing the adjusted tick into the LongPendingAction struct and then use this for validations.
Comments / Resolution	

Issue_36	stETH rebase can be frontrun to profit from price increase
Severity	**Low**
Description	The stETH token rebases daily which then increases the price of wstETH. Users can frontrun this rebase and profit from a slight price increase immediately after the position has been opened. The fact that this price increase is negligible makes this issue only a low severity.
Recommendations	We do not recommend a change here. This issue should be kept in mind.
Comments / Resolution	

Issue_37	Redundant practice for maxLeverage excess
Severity	**Informational**
Description	Within the position validation logic, an extra scenario has been incorporated in the case where the initiated position exceeds the max leverage of 10 due to a price decrease or positive funding application. This logic seems to be completely redundant as a validated position can anyways exceed the 10x leverage. Moreover it goes without saying that this introduces a lot of redundant complexity which can result in other issues.
Recommendations	At Bailsec, we are of the opinion that with increased code complexity the risk of vulnerabilities increases exponentially. Therefore, it often makes sense to reconsider certain design choices and eventually replace them with a more simplified approach. Consider removing the maxLeverage scenario.
Comments / Resolution	

Issue_38	Native ETH refund to validator allows for potential reentrancy attacks
Severity	**Informational**
Description	Whenever the caller for a validation is not the validator, it will refund the security deposit in the middle of the transaction via a call to the validator. This allows for reentrancy attacks in case of any faulty states. While we currently could not observe such a state, that doesn't mean it is non-existent. Specifically since the codebase is very large.
Recommendations	Consider being careful in the further development and considering this scenario. A security safeguard could be to only transfer WETH instead of ETH.
Comments / Resolution	

ActionsVault

The ActionsVault contract forms the logic contract for vault interactions. Users can therefore deposit wstETH and receive USDN, vice-versa, they can redeem USDN and receive the corresponding amount of wstETH. The conversion mechanism is similar to that of the known ERC4646 vault, leveraging the **rule of three**:

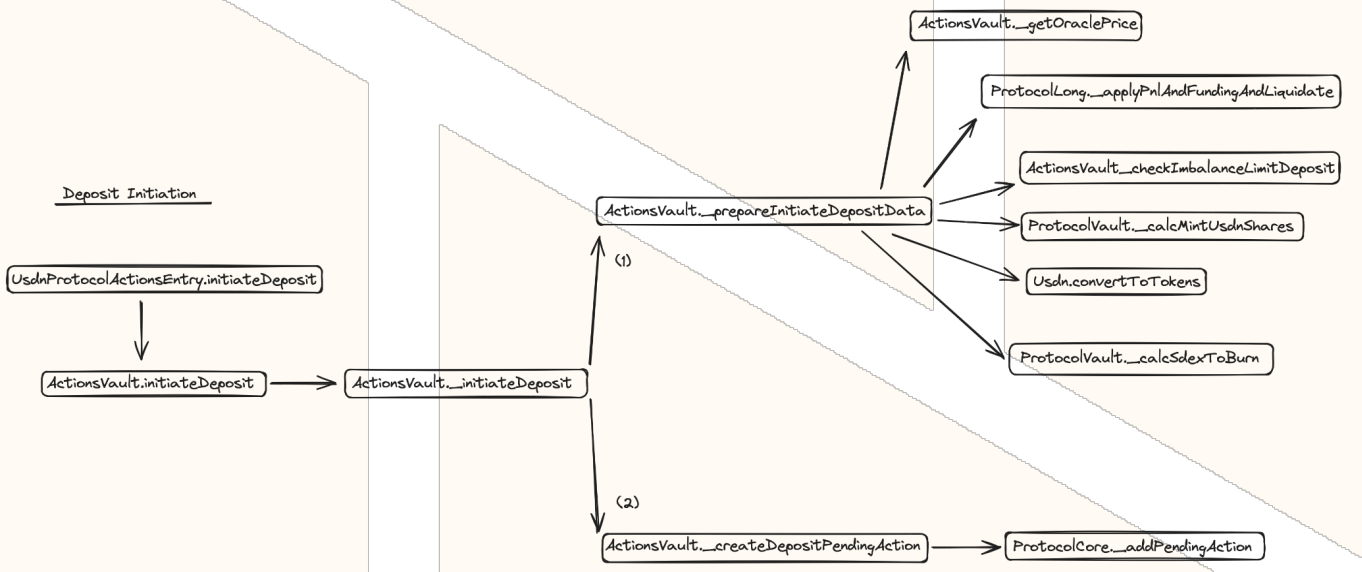
Deposit: wstETHAmount * totalUSDNShares / wstETHAmountInVault

Withdrawal: USDNSharesToRedeem * wstETHAmountInVault / totalUSDNShares

Therefore one can see that USDN is solely backed by wstETH instead of USD. This fact forms the main difference between a USD-backed stablecoin and USDN.

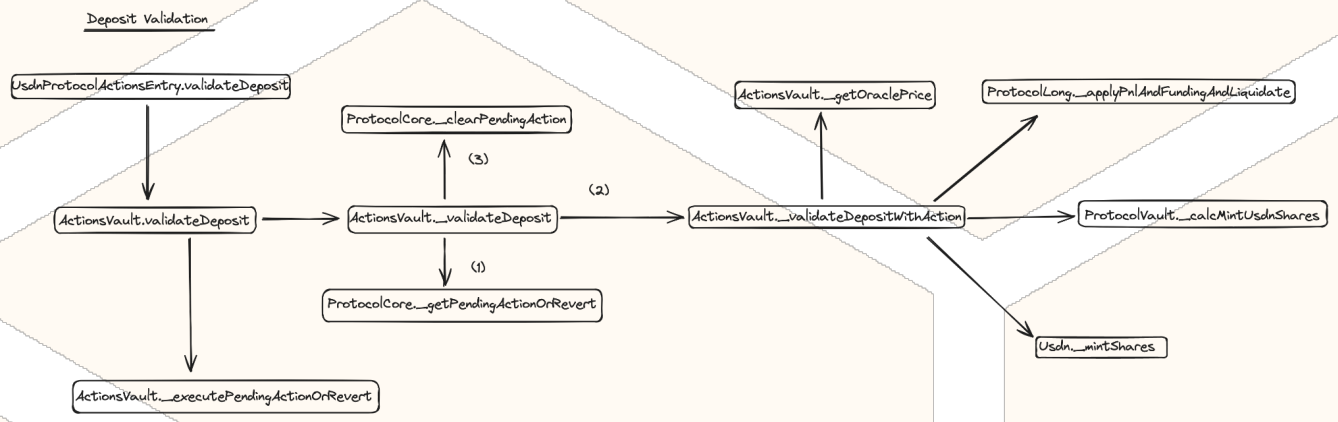
Deposit Initiation: This function allows users to initiate a deposit by providing ETH to then receive USDN once the initiation is validated. Additionally, users will pay a small fee on their deposit which means they will receive slightly less USDN, furthermore, a specific amount of SDEX must be provided which is then sent to the dead address.

Whenever a deposit action has been successfully created, it is added as a pending action.

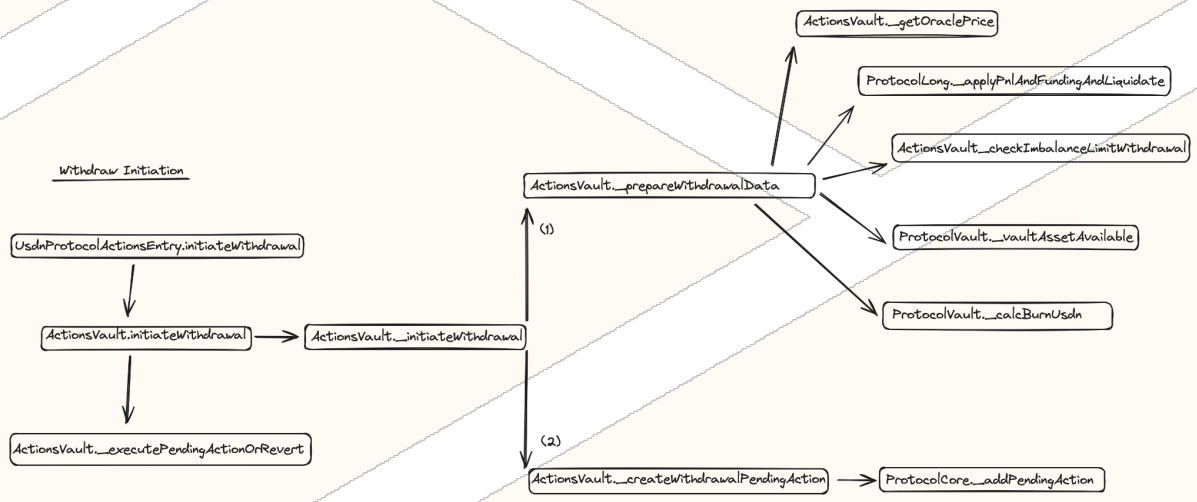


Deposit Validation: Once a deposit is initiated, it can be validated after 24 seconds by any address. The security deposit is however always transferred to the validator, as long as the validation is done within the first 90 minutes. If the validation is not done within 90 minutes, anyone can pick up the security deposit by validating the actionable action. Upon validation, the action is picked and removed from the queue, the contract state is updated and the corresponding USDN shares are minted to the recipient.

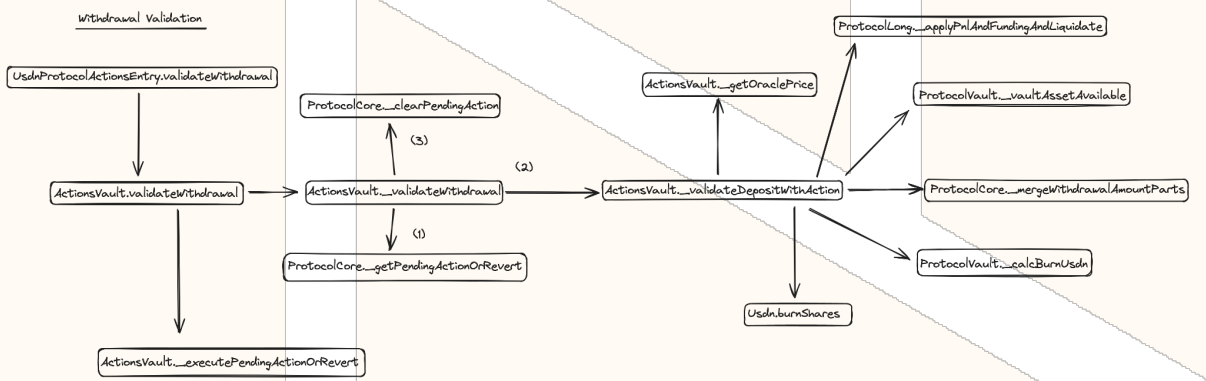
The only real impact of the validation is the fact that potentially a new price is being used to calculate _balanceVault. However, if this new price is in favor of the user, the initial price will be used, resulting in less USDN for the user.



Withdraw Initiation: Any user with a stake can initiate a withdrawal which requires the corresponding amount of USDN to be transferred to the contract and creates a pending action. Upon each withdrawal, a fee is applied which decreases the received wstETH amount. Any initiated position can be validated after 24 seconds in the same manner as the deposit validation.



Withdraw Validation: Once a withdrawal is initiated, it can be validated after 24 seconds by any address. The security deposit is however always transferred to the validator, as long as the validation is done within the first 20 minutes. If the validation is not done within 20 minutes, anyone can pick up the security deposit by validating the actionable action. Upon validation, the action is picked and removed from the queue, the contract state is updated and the corresponding wstETH amount is transferred to the recipient.
The only real impact of the validation is the fact that potentially a new price is being used to calculate _balanceVault. However, if this new price is in favor of the user, the initial price will be used.



Furthermore, this library exposes logic to execute actionable actions (>90 minutes after init state)

Appendix: Imbalance checks

The protocol functions in a way that regularly “transfers” assets between the vault and the long side due to the funding rate. Due to several reasons, including but not limited to the fact that each side eventually pays the other funding and PnL and in an effort that the protocol works flawlessly, it is important that both sides remain partially balanced at all times. To achieve this goal, imbalance checks were incorporated. notably during the following interactions:

a) Deposit into the vault
b) Withdraw from the vault
c) Open a long position
d) Close a long position

To understand how these checks work, we will illustrate the imbalance check which is executed upon the vault deposit:

```
(newVaultExpo - currentLongExpo) * 10_000 / currentLongExpo
```

whereas in our example:

newVaultExpo: Total assets in the vault including the new deposit
currentLongExpo: Total long balance (without collateral)

Therefore, the above formula simply calculates how much % the vaultExposure will be higher/lower than the long exposure

Consider the following values:

currentLongExpo = 100e18
vaultExpo = 100e18
deposit = 10e18

following the formula:

(110e18 - 100e18) * 10000 / 100e18 = 1000

Therefore, the imbalance % between the vault and the long side is 10%

A specific scenario is the liquidation where it may be possible that the forced position closure will result in a protocol imbalance. In such a scenario, the protocol implements a counter mechanism (Rebalancer) which opens a position to increase the tradingExpo and bring the protocol in an equilibrium state.

Appendix: Security Deposit Refund:

The Security Deposit Refund mechanism is a fundamental principle in the USDN architecture and ensures incentives to validate any action promptly. This mechanism is exactly the same as within the ActionsLong contract and therefore the Appendix will be added to the ActionsVault contract but remains representative for both contracts/libraries.

Whenever any action (deposit/withdraw/opening/closing) is initiated, users need to provide a security deposit in ETH. The reason for this security deposit is to incentivize the validator to execute this action and receive the security deposit back. Notably, anyone can validate an initiated action after 24 seconds, however, within the first 90 minutes, the security deposit will always be transferred to the validator. After 90 minutes have been surpassed, the validation can be manually executed using another path which then transfers the security deposit to the caller instead of the validator.

The refund amount will be exactly as follows:

msg.value + amountToRefund - securityDepositValue - ethSpent

Appendix: Deposit and Withdrawal Fee

On deposits and withdrawals, a fee is applied which will always result in users receiving less output tokens/shares. This fee is not nominally applied but is incorporated into a theoretical PnL

application.

Fee on deposits: Whenever users deposit wstETH, the following formula is used to calculate how much USDN shares will be received:

receivedUSDNShares = wstETHAmount * totalUSDNShares / _balanceVault

The _balanceVault variable corresponds to the amount of wstETH on the long side. Now to understand the fee mechanism, we need to understand that the fee is not nominally applied to decrease the provided wstETHAmount. Instead the fee is incorporated in a theoretical PnL application.

The pendingActionPrice is determined as follows:

pendingActionPrice = currentPrice - currentPrice * _vaultFeeBps / 10000

As we can see, the result of this calculation is a price which is lower than the real current price. This is owed to two facts:

a) currentPrice is the scaled down price from the oracle
b) The price has been decreased by a fee

This price is then further used to calculate a theoretical PnL and apply it on _balanceVault:

_balanceVault = balanceVault - (tradingExpo*priceDiff/newPrice)

Therefore, the _balanceVault variable is increased, which means that the divisor in the conversion formula increases as well and thus users will receive less USDN:

receivedUSDNShares = wstETHAmount * totalUSDNShares / _balanceVault

If now users want to redeem the received USDN amount, the real _balanceVault variable will be used (without the theoretical PnL), which means they will immediately experience a loss

Fee on withdrawals:

Whenever users redeem USDN, the following formula is used to calculate how much wstETH will be received

USDNSharesToRedeem * _balanceVault / totalUSDNShares

The _balanceVault variable corresponds to the amount of wstETH on the long side. Now to understand the fee mechanism, we need to understand that the fee is not nominally applied to decrease the provided USDN amount. Instead the fee is incorporated in a theoretical PnL application.

The pendingActionPrice is determined as follows:

pendingActionPrice = currentPrice + currentPrice * _vaultFeeBps / 10000

As we can see, the result of this calculation is a price which is higher than the real current price. This is owed to two facts:

a) currentPrice is the scaled up price from the oracle
b) The price has been increased by a fee

This price is then further used to calculate a theoretical PnL and applied on _balanceVault:

_balanceVault = balanceVault - (tradingExpo*priceDiff/newPrice)

Therefore, the _balanceVault variable is decreased, which means that the multiplier in the conversion formula decreases as well and thus users will receive less wstETH:

USDNSharesToRedeem * _balanceVault / totalUSDNShares

Privileged Functions

- none

Issue_39	Deposit fee can result in large loss for users and prevents protocol equilibrium
Severity	**High**
Description	Whenever users deposit, a fee is applied which decreases the newPrice. Based on this newPrice, the balanceVault is determined. This is done using the simple PnL calculation which we are already familiar with: —---------- *Vault._vaultAssetAvailable(deposit.totalExpo, deposit.balanceVault, deposit.balanceLong, priceWithFees, deposit.assetPrice).toUint256(),* *function _vaultAssetAvailable(* *uint256 totalExpo,* *uint256 balanceVault,* *uint256 balanceLong,* *uint128 newPrice,* *uint128 oldPrice* *) public pure returns (int256 available_) {* *int256 totalBalance = balanceLong.toInt256().safeAdd(balanceVault.toInt256());* *int256 newLongBalance = Core._longAssetAvailable(totalExpo, balanceLong, newPrice, oldPrice);* *available_ = totalBalance.safeSub(newLongBalance);* *}* tradingExpo * priceDiff / newPrice —----------—----------

If we now consider a 10% fee and a deposit of 1 ETH, it should result in a depositAmount of 0.9 ETH. However, that is not the case. In reality, this 10% will decrease the price and it will calculate PnL based on 10% decrease and the trading exposure, which then in turn increases balanceVault and thus increases the divisor such that users will receive less USDN. If users now attempt to withdraw, the _balanceVault variable will be lower than the calculated one above (due to the fact that this negative PnL was only theoretically and not practically applied). This means users will receive a specific amount of USDN but this USDN amount will not translate into the same wstETH amount as deposited.

This is not only fundamentally wrong but can result in a full loss of funds if the long exposure is large enough.

Even though the fee is in reality only 0.04%, it can still result in an undesired outcome, therefore we highly recommend applying a strict nominal fee.

One could now argue that this fee is part of the general design to incentivize vault deposits whenever there is a large trading exposure compared to _balanceVault which then brings the protocol in an equilibrium.

However, there is a second point to mention. If we reconsider the mechanism how the deposit fee is applied, we realize the higher the tradingExpo, the higher the PnL impact:

balanceVault - (tradingExpo*priceDiff/newPrice)

If now _balanceVault is a relatively low value, this means that the impact of the PnL will be percentually larger, the lower _balanceVault is.

	Summarized: a) High tradingExpo = high PnL calculation b) Low _balanceVault = high PnL impact Therefore, if these two points are given, the fee will be higher. At the same time if these two points are true that means that the long side is in dominance which should then in reverse incentivize deposits (by potentially applying a smaller fee - not a higher fee). In contrast to this, if we consider the withdraw function, the larger the tradingExpo, the larger the fee, the less tokens users will receive. This logic is perfectly fine and aligns with the contract equilibrium system because users are incentivized to not withdraw if the tradingExpo (in relation to _balanceVault) is high.
Recommendations	There are two possible ways to move forward: a) Apply a nominal fee upon deposits. This is likely not desired as a design choice. b) Reverse the fee logic to ensure deposits are incentivized whenever the tradingExpo in relation to _balanceVault is high, which supports the contract equilibrium goal.
Comments / Resolution	

Issue_40	Initiation without priceData can result in immediate gain for users
Severity	**High**
Description	The general rule for any state transition within the architecture is that the contract state should always be up to date. This is specifically important for PnL and funding applications. If users initiate a deposit, they can provide a priceData parameter, which is however not mandatory to provide. To illustrate this issue, we can consider a simple deposit initiation and its corresponding oracle fetching mechanism. Upon this action, it will always invoke the _getInitiateActionPrice function. The _getInitiateActionPrice function has several different outcomes, whereas for most scenarios, not the exact current timestamp is returned. In the scenario of a valid Pyth price update, it may actually be possible that the current block.timestamp is returned. For all other scenarios, (potentially) not the most recent price is used but a price which is a bit delayed. This depends on the configured allowed delays. In the scenario of such a delay, the returned timestamp will not be the current timestamp. This is now specifically important in how the returned timestamp will be used: *(, data_.isLiquidationPending) = Long._applyPnlAndFundingAndLiquidate(* *s,* *currentPrice.neutralPrice,* *currentPrice.timestamp,* *s._liquidationIteration,*

	false, *ProtocolAction.InitiateDeposit,* *currentPriceData* *);* The marked timestamp is then used to calculate the funding. This means in the worst case scenario, users can theoretically calculate the funding rate (which is to be applied until the most recent timestamp), initiate a deposit while only updating the funding rate with the returned timestamp and then later potentially profit from the funding rate which will be applied **after** the deposit has been validated (the deposit will be created with the init state, even if validated much later). This means that users can exploit the potential discrepancy between the returned timestamp and the actual timestamp and its impact on the funding rate. *This issue can be 1:1 applied on the position openings. Additionally but less impactful is the withdrawal scenario where users can actually avoid the funding application up to the recent timestamp.
Recommendations	Consider making it mandatory for users to initiate a deposit with the most recent block.timestamp as the latest price update.
Comments / Resolution	

Issue_41	Liquidations/funding/PnL will result in more SDEX being burned than anticipated
Severity	**High**
Description	First of all, we need to clarify that liquidations will increase the balance on the vault side and therefore the USDN price increases. If the USDN price is higher, this means a rebase is triggered which will then in turn increase the overall USDN supply. The same may happen as well with funding and PnL (assuming it is in the corresponding direction, of course it can also be in the other direction). ^ is exactly what can and will happen during a deposit initiation. Now since we have acknowledged this fact, we need to take a look at how the burned SDEX amount is being calculated: usdnAmount * sdexBurnRatio / 1e8 Notably in this formula, usdnAmount is not the nominal amount of shares which are received but it is the nominal amount of tokens which are received. In a rebase scenario, the price decreases but the supply increases, **which means it will have no real effect on the nominal share value**. If a rebase has happened, this means that users receive a larger amount of tokens (but the same amount of shares, which is equally the same amount of wstETH without the rebase having happened). However, the amount of SDEX to be burned will increase in that scenario, while users will not have any benefit from the rebase. Additionally to this, there is no slippage parameter which means that users are completely extradited to this blunder.
Recommendations	Consider calculating the SDEX burn amount based on the to received shares and not on the to received amount of tokens and allowing users

	to provide a “maxSDEXBurn” parameter which prevents users from burning more SDEX than initially anticipated. In the scenario where it is a design choice that the burned SDEX amount will be increased whenever a rebase has happened, we recommend to only add a slippage check. However, it should be considered that this will eventually result in less protocol activity with increasing ETH prices and frequent protocol liquidations. Specifically the higher fee will then potentially result in lower vault deposits and will result in further protocol imbalances as the vault side may not be aligned with the long side.
Comments / Resolution	

Issue_42	Depositor will almost lose all wstETH if depositing when _balanceVault = 0
Severity	**High**
Description	The main issue is only valid in case _vaultFeeBPS is zero or tradingExpo is zero (due to fee application logic), the secondary issue can happen under any circumstance **Main Issue** Whenever USDN is being minted, there are two possible scenarios to calculate the amount of USDN shares received: a) _balanceVault != 0 -> wstETHAmount * usdnShares / balanceVault b) _balanceVault = 0 -> wstETHAmount * wstETHPrice / 1e18 The first scenario is meant to be used during the initialization only, however ... While it is highly unlikely that the vault side is completely emptied, it is possible that during the normal business logic the vault side can become zero (due to funding - which is currently flawed - and positive long PnL), while there are still outstanding shares. This case is explicitly handled in different scenarios throughout the architecture, such as here (example): https://github.com/SmarDex-Ecosystem/usdn-contracts/blob/ff2b96fefc6ab41bb2adccef430c1f2f19e29dda/src/UsdnProtocol/libraries/UsdnProtocolLongLibrary.sol#L1147 If a user deposits in that scenario, the user will basically unknowingly donate the deposited wstETH almost in full among all other participants (depending on how much USDN shares are outstanding)

while only receiving a partial of the deposit back.

Secondary Issue:

Furthermore, there is an even more complex edge-case within the _validateDepositWithAction function:

a)

```
uint256 usdnSharesToMint1 = Vault._calcMintUsdnShares(
        s, deposit.amount, deposit.balanceVault,
deposit.usdnTotalShares, deposit.assetPrice
        );
```

b)

```
uint256 usdnSharesToMint2 = Vault._calcMintUsdnShares(
        s,
        deposit.amount,
        // calculate the available balance in the vault side if the price
moves to `priceWithFees`
        Vault._vaultAssetAvailable(
                deposit.totalExpo, deposit.balanceVault,
deposit.balanceLong, priceWithFees, deposit.assetPrice
        ).toUint256(),
        deposit.usdnTotalShares,
        priceWithFees
        );
```

In the scenario where priceWithFees is larger than assetPrice, it is possible for the result of _vaultAssetAvailable to become zero due to the positive PnL and decrease of _balanceVault. Under normal circumstances, this would not expose an issue because that means usdnSharesToMint1 is smaller and thus only this value is considered. However, in the scenario where _balanceVault becomes zero, the

	following conversion rate is used: wstETHAmount * wstETHPrice / 1e18 which then potentially results in a lower usdnSharesToMint2 amount and a large loss for the depositor (also due to the fact that the divisor is likely lower than 1e18 by now: s._usdn.convertToShares
Recommendations	While it is not possible in that scenario to use the standard share calculation due to a division by zero error, the user should theoretically still receive the equivalent amount of shares for his deposit. This is however mathematically impossible due to the fact that there are outstanding shares. Therefore, there is currently no easy fix for this scenario. A possibility would be to implement a safeguard to not allow depositing if _balanceVault is zero which then forces users to wait until the funding mechanism transfers assets from the long side to the vault side, allowing users again to safely deposit using the rule of three: wstETHAmount * usdnShares / balanceVault The secondary issue is easier to fix as in the case where priceWithFees > deposit.assetPrice, the calculation for usdnSharesToMint2 can be simply disregarded.
Comments / Resolution	

Issue_43	_validateWithdrawalWithAction and _validateDepositWithAction can result in DoS if new price is higher than init price
Severity	**High**
Description	The _validateWithdrawalWithAction function invokes the _vaultAssetAvailable function as follows: *uint256 available2 = Vault._vaultAssetAvailable(* *withdrawal.totalExpo,* *withdrawal.balanceVault,* *withdrawal.balanceLong,* *withdrawalPriceWithFees,* *withdrawal.assetPrice* *).toUint256();* If withdrawalPriceWithFees is larger than withdrawal.assetPrice, it is possible that the return value becomes negative (if _balanceVault is already a very low value): *function _vaultAssetAvailable(* *uint256 totalExpo,* *uint256 balanceVault,* *uint256 balanceLong,* *uint128 newPrice,* *uint128 oldPrice* *) public pure returns (int256 available_) {* *int256 totalBalance =* *balanceLong.toInt256().safeAdd(balanceVault.toInt256());* *int256 newLongBalance = Core._longAssetAvailable(totalExpo,* *balanceLong, newPrice, oldPrice);* *available_ = totalBalance.safeSub(newLongBalance);* *}* This will always revert due to the marked safeCast

	The same will apply to the _validateDepositWithAction function. If we take a look at the previewWithdraw function, such a case is explicitly handled: *function previewWithdraw(Storage storage s, uint256 usdnShares, uint256 price, uint128 timestamp)* *public* *view* *returns (uint256 assetExpected_)* *{* *// apply fees on price* *uint128 withdrawalPriceWithFees = (price + price * s._vaultFeeBps / Constants.BPS_DIVISOR).toUint128();* *int256 available = vaultAssetAvailableWithFunding(s, withdrawalPriceWithFees, timestamp);* *if (available < 0) {* *return 0;* *}* *assetExpected_ = _calcBurnUsdn(usdnShares, uint256(available), s._usdn.totalShares());* *}* However, in practice this will revert.
Recommendations	For the _validateDepositWithAction function, a fix is trivial: Simply do not consider the priceWithFees if it is larger, as this would anyways result in usdnSharesToMint1 being taken. For the _validateWithdrawalWithAction function a fix is not trivial because it is expected that withdrawalPriceWithFees is larger. In that scenario, we simply recommend to set available2 to zero, as users would not receive any wstETH anyways.
Comments / Resolution	

Issue_44	Lack of slippage check during deposit validation may result in large loss for depositors
Severity	**Medium**
Description	Whenever users deposit wstETH, they will receive the corresponding amount of USDN tokens via the following formula: depositAmount * usdnTotalShares / balanceVault During the deposit initiation, no USDN tokens are yet minted, this only happens during the deposit validation. It is important to understand that during the validation, not the initially calculated USDN amount is minted but potentially a smaller amount which is calculated as follows *uint256 usdnSharesToMint2 = Vault._calcMintUsdnShares(* *s,* *deposit.amount,* *// calculate the available balance in the vault side if the price moves to `priceWithFees`* *Vault._vaultAssetAvailable(* *deposit.totalExpo, deposit.balanceVault, deposit.balanceLong, priceWithFees, deposit.assetPrice* *).toUint256(),* *deposit.usdnTotalShares,* *priceWithFees* *);* It is furthermore important to understand that the smaller of both calculation outputs is used. If any large price swings are happening, this can result in an unexpected loss/gain for a user, more specifically if more than 20 minutes have been passed as this will then fetch the CL price
Recommendations	Consider implementing a slippage functionality which in the worst

	case simply reverses the deposit initiation.
Comments / Resolution	

Issue_45	Rare edge-case due to price arbitrage may allow users to extract value from the protocol due to “due liquidations”
Severity	**Medium**
Description	Whenever users initiate a deposit, the initiated contract state is being used to calculate the received USDN amount. Upon validation, the only thing which changes is the price and therefore eventually the _balanceVault variable. Though, _balanceVault will still be based on the initial state at this point: *uint256 usdnSharesToMint2 = Vault._calcMintUsdnShares(* *s,* *deposit.amount,* *// calculate the available balance in the vault side if the price moves to `priceWithFees`* *Vault._vaultAssetAvailable(* *deposit.totalExpo, deposit.balanceVault, deposit.balanceLong, priceWithFees, deposit.assetPrice* *).toUint256(),* *deposit.usdnTotalShares,* *priceWithFees* *);* Now, as we have already explained multiple times, users will have some arbitrage possibility in the price fetching mechanism because they can eventually use the outdated chainlink price. The Smardex team was aware of the fact that this is possible and this is exactly the reason why the initiate/validate mechanism was incorporated (to

	counter exploits where users open long positions with an outdated price and then benefit from it). Unfortunately, the fact that liquidations are based on the returned price, are not considered into this mechanism. This means if a user spots an outdated CL price combined with an outdated Pyth price, the user can initiate a deposit with this outdated price, then validate that deposit and then profit from eventual liquidations which should have happened at the time of the deposit but only happen afterwards (due to using the outdated price). The reason why this issue was only rated as medium is because it is very rare to happen that the price deviation between the real price and the *worst* oracle price is large enough to bypass a due liquidation.
Recommendations	Consider forcing users to update and use the most recent Pyth price during initiations.
Comments / Resolution	

Issue_46	Lack of exclusivity for validator validation can result in worse outcome
Severity	**Medium**
Description	As we have already outlined within the OracleMiddleware section, there are two possibilities to fetch the price within the first 20 minutes after the initiation, which is either the Redstone case or the Pyth case. Now we need to understand that, while the validator will receive the 0.5 ETH within the first 90 minutes, anyone can validate an initiated action. This means also that anyone can provide the priceData parameter which is responsible for fetching the price and timestamp. A malicious user can therefore inspect which oracle has the least favorable price for the initiated action and can then use a carefully crafted priceData to fetch this price, which then results in an opportunity loss for the user.
Recommendations	Consider either not allowing users to validate any initiated action or only allow to consider one price oracle.
Comments / Resolution	

Issue_47	Validator can withhold validations for up to 90 minutes
Severity	**Medium**
Description	As we have already explained, within the first 90 minutes, the security deposit value is transferred exclusively to the validator. The ETH transfer is happening as follows: *(bool success,) = to.call{ value: amount }("");* This will not work if the validator is a smart contract that disables the receipt of ETH. While it is clear that the validator is assumed to be a trusted address, there is still the possibility for a validator to become malicious and withhold the validation for up to 90 minutes, while then in the last block(s) validating the action to still get the allocated ETH amount. An initiated action which is more than 20 minutes in the initiated state, will always consult the Chainlink price after the targetLimit, which is not as expected init TS + 24 seconds but init TS + 20 minutes: *uint128 targetLimit = targetTimestamp + _lowLatencyDelay;* This can result in a disadvantage for the user that has initiated the action.
Recommendations	Consider wrapping ETH to WETH and transfer out WETH if the low-level call reverts.
Comments / Resolution	

Issue_48	Stale deposit/withdraw initiation can result in largely inaccurate conversion rate
Severity	**Low**
Description	Whenever users deposit wstETH, they will receive USDN via the following conversion: amount * usdnTotalShares / balanceVault For withdrawals, the calculation is as follows: usdnToBurn * balanceVault / usdnTotalShares So far, these calculations are totally fine. It is just important to point out that a fee upon these interactions is applied which will either increase/decrease balanceVault, towards the favor of the protocol such that users will receive less tokens. During the validation phase, this fee is applied on the currentPrice and then again favors the protocol: The less desired outcome for users (compared the initiate and validate price) is used. However, when calculating these outcomes, the init state of the protocol is used, example: *uint256 available2 = Vault._vaultAssetAvailable(* *withdrawal.totalExpo,* *withdrawal.balanceVault,* *withdrawal.balanceLong,* *withdrawalPriceWithFees,* *withdrawal.assetPrice* *).toUint256();* This means, any change in the vault state will not be reflected, which can highly alter the outcome if the validation does not immediately

	happen after 2 blocks. This issue is only rated as low severity because it was communicated by the protocol that this is a design choice.
Recommendations	Consider if this issue is acceptable, if no, it is required to refactor this logic.
Comments / Resolution	

Issue_49	Direction is not considered for CL price
Severity	**Low**
Description	Whenever a deposit/withdrawal is being validated, there are several different scenarios of how the price can be fetched: a) Pyth b) Redstone c) Chainlink For Pyth and Redstone, the direction is considered, for Chainlink this is however not the case. Fortunately, it is only possible to use the CL oracle 20 min after the initiation, therefore this issue can be considered as low severity.
Recommendations	Consider either acknowledging this issue or implement a similar direction rounding as with the other oracles.
Comments / Resolution	

Issue_50	SDEX fee calculation does not round up
Severity	**Low**
Description	Whenever fees are taken, these should round up, against the favor of the user. As we have already raised an issue for the SDEX fee calculation: *"Liquidations will result in more SDEX being burned than anticipated"*, this issue can be completely disregarded if the other one is fixed. However, currently the amount of SDEX to be burned is calculated based on the token amount which is derived by the convertToTokens function within the Usdn contract. This function does not round up but rounds to the nearest integer.
Recommendations	Consider disregarding this issue by fixing *"Liquidations will result in more SDEX being burned than anticipated"* or use the convertToTokensUp function within the USDN contract.
Comments / Resolution	

Issue_51	Native ETH refund to validator allows for potential reentrancy attacks
Severity	**Informational**
Description	Whenever the caller for a validation is not the validator, it will refund the security deposit in the middle of the transaction via a low-level call to the validator. This allows for reentrancy attacks in case of any faulty states. While we currently could not observe such a state, that doesn't mean it is non-existent. Specifically since the codebase is very large.
Recommendations	Consider being careful in the further development and considering this scenario. A security safeguard could be to only transfer WETH instead of ETH.
Comments / Resolution	

Issue_52	Violation of checks-effects-interactions pattern
Severity	**Informational**
Description	Within the _checkPendingFee function, the CEI pattern is violated due to the transfer before the storage change. To ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes).
Recommendations	Consider following the CEI pattern: https://bailsec.io/tpost/gxcih1xoy1-checks-effects-interactions
Comments / Resolution	

ActionsUtils

The ActionsUtils library is an utility library which exposes functionality for many different purposes, it is leveraged throughout the long opening and closure process via the following functions, below we will enumerate each helper function and its corresponding task:

a) _createOpenPendingAction: Creates the pending action for position opening and adds it to the queue

b) _prepareValidateOpenPositionData: Helper function for opening validation

c) _removeAmountFromPosition: Helper function that fully or partially removes a position from the system

d) _saveNewPosition: Helper function that adds a new position to the system

e) _prepareClosePositionData: Helper function to initiate a position closure

f) _createClosePendingAction: Creates the pending action for position closure and adds it to the queue

g) _checkInitiateClosePosition: Helper function that takes care of validations for position closure initiation

h) _assetToRemove: Helper function that calculates how much from a position should be removed.

Furthermore this library allows to change the owner of a position and facilitates the entry for liquidations, as well as transfers out liquidation rewards and allows for executing actionable actions.

Appendix: Position storage at tickHash

Whenever a new long position is opened, it is saved under a specific tickHash. The tickHash is derived from the tick and the current version of the corresponding tick. By default the version for each tick is 0 and it is incremented by 1 whenever a tick is liquidated.

The tickHash is determined as: keccak256(abi.encodePacked(tick, version))

Privileged Functions

- none

Issue_53	Position openings with fee will alter unadjusted liquidation price instead of increasing _vaultBalance
Severity	**High**
Description	First of all, we need to quickly re-iterate the idea behind unadjusted and adjusted prices. Whenever a position is opened, the desiredLiqPrice will be converted to the unadjustedPrice using the following formula: desiredLiqPrice * accumulator / (currentPrice * tradingExpo) This means the position is exactly stored at the unadjustedPrice for the desiredLiqPrice (whatever that now might be is irrelevant at this point). It is just important to understand if we now liquidate positions, the currentPrice which is the real price is also unadjusted, following the same formula desiredLiqPrice * accumulator / (currentPrice * tradingExpo) This means, liquidations completely happen based on the unadjusted base, so to speak. If we now have a currentPrice of 3000 and this price converts to an

unadjustedPrice of 3000 but after some changes it converts to an unadjustedPrice of 3300, this means positions are liquidated *later*, because only positions above 3330 are liquidated. In reverse this means that the collateral for all long positions has been increased.

We will use this statement to prove that position openings with a fee do not increase the vault balance but will flow into the collateral of all existing long balances (the liquidation price will increase, as described above). Notably, as per Smardex team, it is desired that this fee flows towards the vault side.

Status Quo:

The contract is initialized by governance with the following parameters:

depositAmount = 10e18
longAmount = 1e18
desiredLiqPrice = 2754e18
currentPrice = 3000e18

This will give us the following settings:

position.tick = 79014
position.totalExpo = 10e18
_balanceVault = 10e18
_balanceLong = 1e18
_totalExpo = 10e18
currentPrice = 3000e18
_liqMultiplierAccumulator = 2.7e40
_tradingExpo = 9e18

Consider the current unadjusted liquidation price for long positions is 3000 for the current price of 3000

	unadjustedPrice(3000e18) = 3000e18 * 2.7e40 / (3000e18*9e18) -> 3000e18 a) A position is opened with the following parameters > amount = 1e18 > desiredLiqPrice = 2754e18 (10x) > 10% fee b) This will result in the following settings > adjustedPrice = 3030e18 > position.tick = 79212 > liqPriceWithoutPenalty = 2700e18 > position.totalExpo = 5.5e18 > _balanceLong = 2e18 > _totalExpo = 15.5e18 > accumulator = 4.185e40 c) If we now determine the new unadjusted price for currentPrice = 3000, this should not change because the fee must flow to the vault side and should not impact other positions. However, see the following calculation: unadjustedPrice(3000e18) = 3000e18 * 4.185e40 / (3000e18*13.5e18) -> 3100e18 As elaborated above, the liquidation price has increased.
Recommendations	Ideally, the following steps will be taken: a) Calculate the fee which is actually deducted from the position (pos.tradingExpo * priceDiff)

	b) Deduct this fee from the position amount c) Following with all calculations using the real price, not the price with the fee.
Comments / Resolution	

Issue_54	Rebalance withdrawal case incorrectly assumes “to” address as position owner
Severity	**Medium**
Description	Within the _checkInitiateClosePosition, the following special scenario is incorporated: *// make sure the remaining position is higher than _minLongPosition* *// for the Rebalancer, we allow users to close their position fully in every case* *uint128 remainingAmount = pos.amount - amountToClose;* *if (remainingAmount > 0 && remainingAmount < s._minLongPosition) {* *IBaseRebalancer rebalancer = s._rebalancer;* *if (owner == address(rebalancer)) {* *uint128 userPosAmount = rebalancer.getUserDepositData(to).amount;* *if (amountToClose != userPosAmount) {* *revert IUsdnProtocolErrors.UsdnProtocolLongPositionTooSmall();* *}* *} else {* *revert*

	IUsdnProtocolErrors.UsdnProtocolLongPositionTooSmall(); *}* *}* This means, if a position is from the rebalancer, it can be fully withdrawn. However, the "to" address only reflects the recipient of the tokens and not the corresponding msg.sender in the userDeposits[address] mapping. Thus in this special scenario it will not work if tokens are transferred to a different recipient than the original position owner.
Recommendations	Consider using the position owner instead of the recipient.
Comments / Resolution	

Issue_55	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 *A similar issue is existing for position openings
Recommendations	Consider incorporating the fee into the imbalance check.
Comments / Resolution	

Issue_56	Lack of liquidation incentive if _balanceVault is near zero
Severity	**Low**
Description	The _sendRewardsToLiquidator function transfers liquidation rewards to the corresponding liquidator. These rewards are taken from the vault side but in the scenario where the vault has insufficient funds, it is possible that not the full amount of liquidation rewards are transferred to the liquidator. While this issue is unlikely due to the fact that previously a position must have been liquidated (and thus the remaining collateral has been allocated to _balanceVault), it is not completely unthinkable (if for example the liquidated position has no leftover value). This means, liquidators might not be sufficiently incentivized to liquidate users.
Recommendations	Consider following a different approach and outsourcing funds for liquidations.
Comments / Resolution	

Issue_57	Closure imbalance check may result in larger tradingExpo if position leverage is <1
Severity	**Low**
Description	As we have already elaborated in other issues, it is possible for a position to have a leverage which is below 1. In such a scenario, the position closure would then further imbalance the protocol towards the vault side, which remains unchecked during the _checkImbalanceLimitClose function.
Recommendations	Consider ensuring that it is not possible for the position leverage to become < 1.
Comments / Resolution	

Issue_58	Liquidation rewards will be taken from the vault
Severity	**Low**
Description	Currently, liquidation rewards are sent out straight from the vault side to the liquidator, while this does not have an immediate negative side-effect, it will quickly become an issue if liquidations get exploited and thus it allows the whole vault side to be drained.
Recommendations	Consider implementing a safety module which stores a reasonable amount of wstETH and only transfer liquidation rewards from this module. This could be done by taking an additional nominal fee on vault deposits combined with manually monitoring the safety module balance, ensuring that it always remains sufficiently covered.
Comments / Resolution	

ProtocolCore

The ProtocolCore contract is a fundamental part of the overall architecture and 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.

Appendix: Queue setting:

There are several different scenarios where an action is added/removed to/from the queue.

a) Addition to the queue: Whenever a new action is initiated, the initiated action is pushed at the end of the queue.

b) Removal from the queue: There are several different scenarios when initiate actions are removed from the queue.

1) Standard validation: The standard validation happens if any address validates an initiated action within the first 90 minutes. The initiated action is removed or cleared out from the queue, in the latter case, it may still exist in the queue but will be empty and once the queue has reached the corresponding index, it will automatically be removed.

2) After-action-validation: Whenever an action has happened (initiation or validation), the protocol always attempts to validate any actionable actions which are more than 90 minutes in the init state. The initiated action is removed or cleared out from the queue.

3) Actionable validation: Whenever an action is pending for more than 90 minutes, it can be directly validated by any user. The initiated action is removed or cleared out from the queue. This follows the same logic as in 2).

4) Liquidated action: Whenever an action is already liquidated, it will be removed from the queue when a new action for the corresponding validator is being added or whenever the action gets validated.

5) Blocked action: Whenever an action is considered as blocked, which is 2.5 hours after the initiation, it can be manually removed from governance.

Appendix: _totalExpo & tradingExpo

Whenever a new position is opened, this will impact the _totalExpo and _balanceLong variables. Assume a user opens a long position with 1 ETH and 5x leverage, this will set these variables as follows:

_balanceLong = 1e18
_totalExpo = 5e18

The calculation for the total exposure is as follows:

amount * startPrice / (startPrice - liquidationPrice)

To determine the tradingExpo/longExpo, the calculation is as follows:

_totalExpo - _balanceLong

> 5e18 - 1e18 = 4e18

Appendix: PnL calculation & Liquidation impact

Whenever the protocol state is updated (_applyPnLAndFundingAndLiquidate), this will calculate the PnL from all long positions since the last update time. There are three possible scenarios:

a) The price has not changed: No changes will be made.

b) The price has increased: Long positions have gained a profit. The long side will increase and the vault side will decrease.

c) The price has decreased: Long positions encounter a loss. The long side will decrease and the vault side will increase.

The formula to calculate PnL is as follows:

tradingExpo * priceDiff / newPrice

Whenever the contract state is updated, this will calculate the PnL since the last time and apply it on _longBalance and _vaultBalance. Since this will change the _longBalance, it will affect the collateral of all users for their long position (increase/decrease).

As per Smardex team, any PnL distribution should alter the position value but not the liquidation price of a position. Even though the PnL application may increase the provided collateral of a position and therefore decrease the leverage, it is not intended to adjust the liquidation price of this position. This is because of their unique implementation of how positions will be calculated.

Below can be found an illustration of the PnL and corresponding position value:

- price = 3000
- amount = 1 ETH
- leverage = 10x
- tradingExpo = 9 ETH
- totalExpo = 10 ETH

a) The price increases by 10%
- price = 3300

- overall value = 33000
- borrowed value = 27000
- position value = 6000
- position value in ETH = 1.818 ETH

b) The price increases by 100000%
- price = 330000
- overall value = 3 300 000
- borrowed value = 27000
- position value = 3 273 000
- position value in ETH = 9.9191 ETH

This aligns with the formula how the position value is actually calculated:

> position.totalExpo * (currentPrice - liquidationPriceWithoutPenaltyAdjusted) / currentPrice

a) 10e18 * (3300e18 - 2700e18) / 3300e18 = 1.818 ETH
b) 10e18 * (330000e18 - 2700e18) / 330000e18 = 9.918 ETH

Appendix: Funding rate distribution

The funding rate is a fundamental instrument in the USDN ecosystem. It aims to balance vault deposits and long exposure. If the funding rate is negative, the vault will pay the long side. If the funding rate is positive, vice-versa.

The funding rate is distributed periodically, whenever the contract state is updated via the _applyPnlAndFundingAndLiquidate function. In storage adjustment, it is expressed by either decreasing _balanceLong and increasing _balanceVault, or vice-versa. No matter how the funding rate changes, the total exposure of all positions still remains the same. Moreover, the funding fee impacts the liquidation price, as a positive funding fee decreases the global liquidation price and a negative funding fee increases the liquidation price. This is handled in the price unadjustment logic:

> desiredLiqPrice * accumulator / (currentPrice * tradingExpo)

Since a positive funding fee decreases _balanceLong but _totalExpo remains constant, this means that tradingExpo will become larger, with the effect that the unadjusted price becomes smaller and positions will be liquidated earlier. The exact opposite happens in the vice-versa scenario.

Additionally, we will demonstrate the impact of the funding fee on the position value, which is calculated as follows:

> liquidationPriceNoPenaltyAdjusted = liqPriceNoPenaltyUnadjusted * (currentPrice * tradingExpo) / accumulator

> position.totalExpo * (currentPrice - liquidationPriceNoPenaltyAdjusted) / currentPrice

Whenever the funding fee is positive, this will decrease _balanceLong and therefore increase tradingExpo. With an increase of tradingExpo, the liquidationPriceNoPenaltyAdjusted will become higher, thus decreasing the position value. In reverse, an increase of _balanceLong will decrease tradingExpo, which then decreases liquidationPriceNoPenaltyAdjusted and decreases the position value.

It is additionally worth mentioning that upon each funding application, a protocol fee is taken based on the funding rate. This protocol fee is then simply “deducted” from the profit gaining side. If for example the funding fee is positive, it will deduct 10 wStETH from the long side and add 9 wStETH to the vault side, while collecting 1 wStETH as protocol fee.

Appendix: BitMap Index

Each tick has its corresponding BitMap index which is calculated as follows:

(tick - MIN_TICK) / tickSpacing

The idea of this logic is to assign one unique index to each tick, ensuring tickSpacing is considered. This means the minimum populatable tick should have index 0 and each subsequent index (increased by tickSpacing) should have an index as : previousIndex + 1

This index is then used within Solady's LibBitmap library to set and unset ticks whenever they have corresponding long positions.

This Bitmap is then leveraged upon liquidations as from the _highestPopulatedTick downwards, all indexes and corresponding ticks are fetched until the a tick is below the liquidation tick. Fetching from the Bitmap works as follows:

1. Calculate the Bitmap index from the _highestPopulatedTick via _calcBitmapIndexFromTick

2. Fetch the last set index from the Bitmap based on the previous calculated Bitmap index via findLastSet

3. Convert the fetched index to the corresponding tick via _calcTickFromBitmapIndex

This process ensures that under all circumstances all "set" ticks in the Bitmap below and at the _highestPopulatedTick are considered.

Privileged Functions

- none

Issue_59	First depositor has no leverage limitation
Severity	**High**
Description	Currently, anyone can invoke the initialize function to set up the initial contract state. While that in itself does not expose a large problem, this flow lacks a maximum leverage check, which then allows the first depositor to create a long position with unlimited leverage.
Recommendations	Consider checking for the maximum allowed leverage.
Comments / Resolution	

Issue_60	EMA calculation is incorrect
Severity	**High**
Description	The calcEMA function in UsdnProtocolCoreLibrary is calculates the EMA with the formula (lastFunding + previousEMA * (emaPeriod - secondsElapsed)) / emaPeriod which is incorrect. The correct formula is: *currentEMA = alpha * lastFunding + (1-alpha) * lastEMA* *With alpha = 1 - exp(secondsElapsed / emaPeriod)* *(with secondsElapsed <= emaPeriod)* If the goal of this function was simply to return the moving weighted average of the funding, then by fixing the parenthesis, it would also work: *lastFunding + (previousEMA * (emaPeriod - secondsElapsed)) / emaPeriod*

	Here's a numerical example of a clearly incorrect output: If secondsElasped == emaPeriod , it simply will return lastFunding. However, let's say secondsElapsed == emaPeriod - 1 which is only 1 second earlier. It will return (lastFunding + previousEMA * 1) / emaPeriod. If emaPeriod is 60 seconds, then this is 1/30 of the average between lastFunding and previousEMA which would be 30x lower than the value 1 second later if EMA and lastFunding values are close to each other.
Recommendations	A more correct formula would be currentEMA = (lastFunding * emaPeriod) + (lastEMA * (1 - period). However there are other problems with this fixed formula which are described in other issues in the report.
Comments / Resolution	

Issue_61	EMA incorrectly uses funding from wrong time period
Severity	**High**
Description	The call to updateEMA occurs before funding is calculated. Additionally _funding is calculated using oldVaultExpo so this value represents the funding rate from past to current block.timestamp. This funding value is subsequently stored in s._lastFundingRate. Therefore when EMA is updated in the future, it is based on outdated funding rate. Additionally the elapsedSeconds that this funding rate is multiplied by corresponds to a different funding period, which means that funding rates are weighted by the incorrect duration when being incorporated into the EMA.
Recommendations	Consider refactoring the EMA calculation.
Comments / Resolution	

Issue_62	_funding calculation is incorrect
Severity	**High**
Description	The _funding function is responsible for calculating the funding rate. This calculation should incorporate the past EMA of the funding rate and then multiply it with the current funding rate in the known manner while using the multiplier. The calculation however just adds the past EMA to the funding rate. This behavior is incorrect and does not follow the above described funding rate calculation using the EMA.
Recommendations	Consider refactoring the whole calculation and follow the EMA pattern to calculate the updated funding rate.
Comments / Resolution	

Issue_63	The Funding formula breaks down with different times between state updates
Severity	**High**
Description	There was a separate issue that the EMA had problems with disparate time differences between state updates. However, the _funding formula has a similar problem (even if we assume EMA has no problem). Over a very short time, any user could make charging the protocol EMA (fundRate*time) worth of funding every single block by making any state update every block. The EMA(fundRate*time) was calculated over times much longer than a block so this would massively overcharge funding. For example if the EMA was averagely calculated over 2 minutes, and each block is 12 seconds apart, then 10x the correct funding is charged. While it is true that the EMA will eventually decrease, this can be done for many blocks in a row because the change to EMA is time weighted.
Recommendations	Consider refactoring the whole calculation.
Comments / Resolution	

Issue_64	Incorrect storage of _lastFunding and inconsistency in determination
Severity	**High**
Description	Within the _applyPnlAndFunding function, the _lastFunding state variable is stored as follows: *s._lastFunding = fundWithFee;* If we now investigate how fundWithFee is determined, we will get the following different results: a) In the scenario that the funding rate is positive, fundWithFee becomes the nominal funding rate (fund): *fundWithFee_ = fund;* b) In the scenario that the funding rate is negative, fundWithFee will be set as follows: *fundWithFee_ = fund;* *fundWithFee_ -= fund * protocolFeeBps / int256(Constants.BPS_DIVISOR);* As we can see, in scenario b), the funding rate is then *further decreased*, which means the fee is incorporated on top of it, making the funding rate even more negative. The problem: fundWithFee is then stored as _lastFunding, while _lastFunding should in fact just be the nominal funding rate. This is because calcEMA() is used to calculate the funding rate before the fee. Therefore the EMA of the pre-fee funding rate should be used in the calculation rather than the funding rate with fee.

	Therefore, we have multiple issues, such as an inconsistency in scenario a) and b) where the fee is only applied upon b) and the issue itself that the fee should likely not be used to store the last funding rate.
Recommendations	Consider refactoring the whole calculation.
Comments / Resolution	

Issue_65	Funding rate multiplied by time is incorrectly used for EMA calculation
Severity	**High**
Description	The EMA is meant to smooth out changes in funding rate especially when there are sudden changes in vault balances by averaging out over past funding rates. Given that in practice times between state updates are irregular (it could be 12s, then next period is 1 minute), consider which one is better to get the EMA of: - The funding charged over the time period between these irregular state updates - The funding rate If one state update is 12s apart and the next is 1 minute apart, the value added to the EMA for state change 1 would be 5x lower for the second value than state change 2, which is clearly wrong. If an EMA formula is to be used at all, then a funding rate should be used rather than funding since last update.
Recommendations	Consider refactoring the whole calculation.
Comments / Resolution	

Issue_66	Cleanup for validateOpenPosition will not transfer wstETH back and falsifies tradingExpo
Severity	**High**
Description	The _removedBlockedPendingAction function can be invoked by governance via the UsdnProtocolVault entry contract in an effort to remove positions from the queue which are initiated but not validated. This is possible after 90 minutes with the indication that there is a serious problem in the queue. There are two possibilities to execute this: a) cleanup = true: This will reverse all previous state transitions that have been made during the init actions b) cleanup = false: This will not reverse any state transitions but will just remove the element from the queue Whenever an initiated position opening is being removed with cleanup = true, it will reverse all state transitions but it does not transfer out the previously provided wstETH, resulting in this wstETH being permanently stuck in the contract. Furthermore, it will not decrease _balanceLong but _totalExpo, which has the effect that tradingExpo will be falsified.
Recommendations	Consider decreasing _balanceLong and refund the provided wstETH amount.
Comments / Resolution	

Issue_67	Any initiation can be frontrun and blocked for the cost of 0.5 ETH
Severity	**Medium**
Description	Whenever a deposit is initiated, it is pushed into the following mapping for a validator: *s._pendingActions[user]* Notably, every validator can only have one pending action at a time: *if (s._pendingActions[user] > 0) {* *revert IUsdnProtocolErrors.UsdnProtocolPendingAction();* *}* This can be trivially abused by frontrunning any initation and initiating a position creation with the same validator address, thus blocking the initiation. This will however cost 0.5 ETH because the validator now can validate this created pending action and take the 0.5 ETH. This can specifically become a problem if a user attempts to open a long position at a very specific price, say 2900$/ETH while setting their own address as validator. This griefing attack can now be executed to prevent the user opening the position, resulting in an opportunity loss because the transaction now reverts and will not be fulfilled to the desired price.
Recommendations	Consider implementing a delegation mechanism which allows validators to whitelist addresses to initiate actions on their behalf.
Comments / Resolution	

Issue_68	Griefing: Malicious user can intentionally spam the queue by depositing and withdrawing in a specific manner
Severity	**Low**
Description	The _getActionablePendingAction function is invoked whenever actions are initiated and validated. This function ensures that the queue is constantly updated and initiated actions are validated. Users can intentionally grief this process by depositing and withdrawing in such a manner that the queue is not shrinked but the elements are only zero-d out. This means the loop within _getActionablePendingAction will always continue whenever such a zero-d action is fetched and therefore it is possible that initiation validations can be delayed. This is not a security risk in itself but can become rather annoying if the queue grows large as it takes some time to clear the backlog.
Recommendations	We do not recommend a change. However, this scenario should be kept in mind.
Comments / Resolution	

Issue_69	PnL and funding application can alter imbalance state of the protocol
Severity	**Informational**
Description	A core mechanism of the protocol is the PnL and funding application which is always happening at the beginning of any action. This application can alter the imbalance state of the protocol to bring the protocol out of a balanced state and potentially exceeds the determined imbalance thresholds (either to the long side or to the vault side). In such a scenario, there is no rebalance happening. Fortunately, after some tests, we deemed that the impact is not too large and therefore the future funding rate should be sufficient to rebalance the protocol.
Recommendations	Consider either acknowledging this issue or incorporate a rebalancer check for extreme cases.
Comments / Resolution	

Issue_70	Initiated vault balance is not incorporated in EMA calculation
Severity	**Informational**
Description	Currently, whenever a deposit initiation is happening, this will increase _pendingBalanceVault. Whenever the EMA is now calculated, the _pendingBalanceVault amount will not flow into this calculation. Notably, if long positions are opened, these will already flow into the EMA calculation. It may now be completely legitimate to keep it as-is, if that is a desired design choice. However, due to consistency reasons with regards to long positions we need to point this out as an inconsistency issue.
Recommendations	Consider either staying consistent and incorporating the _pendingBalanceVault into the EMA calculation or simply acknowledge this issue if it is considered as a design choice.
Comments / Resolution	

ProtocolLong

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. 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: 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 rebalancer then opens a long position with leverage. The leverage which is used is solely depending on the position size within the Rebalancer and the target tradingExpo to be reached. The target tradingExpo is calculated as follows:

tradingExpo = vaultBalance * 10 000 / (10 000 + 550)

The goal of this formula is to achieve a protocol state which is slightly favored to the vault side, aiming to achieve the vaultBalance to be 105.5% of the tradingExpo.

Appendix: Rebalancer Tick Calculation:

Here is how the tick will be calculated:

a) Calculate the targetTradingExpo:

> _balanceVault * 10000 / (10000+550)

b) Calculate how much tradingExpo is needed to reach the targetTradingExpo:

> targetTradingExpo - tradingExpo

c) Calculate the highest possible tradingExpo that can be reached based on the current assets of the rebalancer and the maximum leverage:

> pos.expo - pos.amount
> positionAmount * rebalancerMaxLeverage / 1e21 - positionAmount

d) Calculate the lowest possible tradingExpo that can be reached based on the current assets of the rebalancer and the minimum leverage:

> pos.expo - pos.amount
> positionAmount * rebalancerMinLeverage / 1e21 - positionAmount

e) Apply upper/bounder limitations. In our real-world example it is only necessary to handle the scenario where highestUsableTradingExpo is below our needed expo.

f) Calculate the corresponding tick without penalty (unadjusted) for the desired tradingExpo:

> currentPrice * tradingExpo / totalExpo

g) Calculate the expo which will result from that position creation and execute an imbalance check. If the imbalance is more than 5.5% towards the vault, increase liqPriceWithoutPenalty by _tickSpacing. This only applies if the position can cover the targetTradingExpo and should counter rounding errors due to the getEffectiveTickForPrice function.

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: Safety Margin

The protocol incorporates a built-in safety margin check which is applied based on the neutral price and the liquidation price with penalty. It is calculated as follows:

> currentPrice * (10_000 - _safetyMarginBps) / 10-000

This simply ensures that the liquidation price (including penalty) is more than 2% lower as the neutral price.

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

Privileged Functions

- none

Issue_71	Lack of _longBalance decrease during _flashClosePosition will alter _longBalance irreversibly
Severity	**High**
Description	Whenever the _triggerRebalance function is invoked, it will automatically invoke the _flashClosePosition function. Whenever this function is invoked, this means a long position is closed, thus _balanceLong must be decreased. In fact, it is decreased in the cached state but not in memory. Therefore, it will return an incorrect longBalance_ value, inflating the actual long collateral.
Recommendations	Consider decreasing _longBalance after the _flashClosePosition call.
Comments / Resolution	

Issue_72	_minLeverage safeguard is void due to funding incorporation
Severity	**High**
Description	The _minLeverage variable ensures that a position has a minimum leverage. This is in line with the overall logic of the contract architecture that requires open leverage (longExpo) in an effort to work. Currently, the minimum leverage for a position is required to be 1.000000001x. This can quickly turn to 0.999999x if the funding rate results in a long position decrease. (Only measured on the global variables: _totalExpo; _balanceVault). A normal position can never become larger than its expo due to the way how the position value is calculated. There are multiple parts of the codebase which assume that it is theoretically impossible for the balanceLong of a position to exceed the exposure. PnL and funding is added to the balanceLong. Therefore if the positive PnL and funding between two consecutive state updates is above 100%, then the balance will exceed exposure and the calculated leverage will go below 1x. There are multiple ways the contract can break, one of which is explained by this code comment: *"if balanceLong == totalExpo or the long trading expo is negative (theoretically impossible), the PnL is we can't calculate a proper PnL value if the long trading expo is negative because it would invert the sign of the amount"* We will furthermore aggregate instances of this issue in here instead of creating separate issues for each: —----------—---------- **Certain edge-cases will result in a DoS of the _handleNegativeBalances function**

Impact: Low

Within the “Flaw within funding calculation may result in _handleNegativeBalances function to revert” issue, we have already elaborated one specific scenario where the _handleNegativeBalances function inevitably reverts due to a misapplication of funding.

However, this is not the only scenario where this function reverts. below we will enumerate edge-cases where it will revert as well:

a) _longBalance and _vaultBalance are both negative

b) _longBalance is positive and _vaultBalance is (nominally) more negative than the nominal value of _longBalance

c) _vaultBalance is positive and _longBalance is (nominally) more negative than the nominal value of _vaultBalance

—----------—----------

Rebalance will always revert if _balanceLong becomes larger than _totalExpo

Impact: High

As we have already elaborated multiple times throughout the report, it is possible that _balanceLong becomes larger than _tradingExpo.

Some spots in the codebase are adjusted to work with that scenario while some other spots are not adjusted. A rebalance will never work due to the following condition:

if (cache.totalExpo < cache.longBalance) {

revert IUsdnProtocolErrors.UsdnProtocolInvalidLongExpo();

	} —----------—---------- **Edge-case can result in DoS of liquidations** Impact: High Whenever all positions have been liquidated, it will automatically invoke the _triggerRebalancer function with the following safety check: *if (cache.totalExpo < cache.longBalance) {* *revert IUsdnProtocolErrors.UsdnProtocolInvalidLongExpo();* *}* A malicious user can break this check by depositing a large sum with the minimum leverage and then wait until funding increases the _longBalance such that the aggregated _longBalances becomes larger than the totalExpo. However, this exploit needs to have the correct contract circumstances (generally low leverage; high vault deposits; negative funding rate) —----------—----------
Recommendations	There are two solutions: a) Rewriting all scenarios where the contract would break. b) Increasing the minimum leverage with the goal that this issue does not happen in practice. While b) seems not 100% sufficient in theory, it might actually be sufficient in practice because users cannot actively exploit the funding

	application (once the EMA calculation has been fixed).
Comments / Resolution	

Issue_73	Deposits (and Position closures) will never work after a rebalance
Severity	**High**
Description	The Rebalancer is triggered whenever the vault side becomes 6% or more in dominance, with the goal to reach a 5.5% vault dominance. To achieve this goal, the target tradingExpo is calculated as follows: vaultBalance * 10000 / (10000 + 550) If we assume a vaultBalance of 100e18, this will yield us the following targetTradingExpo 100e18 * 10000 / (10000+550) = 94.7e18 If we now calculate the dominance towards the vault side after this targetTradingExpo has been reached: (balanceVault - tradingExpo) * 10000 / tradingExpo (100e18 - 94.7e18) * 10000 / 94.7e18 = 559 If we now consider that a deposit or a position closure is happening after that, we realize that the maximum imbalance towards the vault side must be 500 BPS for deposits and 600 BPS for position closures. This will completely revert for deposits and will revert for medium-sized position closures.

	Effectively, the rebalance interaction will result in a DoS of the protocol.
Recommendations	Consider setting a rebalance limit which actually makes sense, such as calculating a targetTradingExpo which is minimally larger than the vaultBalance.
Comments / Resolution	

Issue_74	Flaw within funding calculation may result in _handleNegativeBalances function to revert
Severity	**High**
Description	The _handleNegativeBalances function is invoked on these two occasions: a) At the beginning of the _liquidatePositions function if the longTradingExpo becomes negative or zero b) At the end of the _liquidatePositions function on all occasions This function does the following: a) If tradingExpo is negative -> decrease balanceVault by tradingExpo -> set tradingExpo to zero b) If balanceVault is negative -> decrease tradingExpo by balanceVault -> set balanceVault to zero Now we need to explain the different scenarios when tradingExpo and

balanceVault can become negative.

tradingExpo can become negative if the PnL increases and the funding is negative, as the PnL increase will bring balanceLong near totalExpo and the funding fee movement will finally bring balanceLong above totalExpo.

balanceVault will become negative if liquidations of underwater positions are happening and the funding rate is negative (though, a negative funding rate means that balanceLong is larger than tradingExpo, thus this alone should not bring balanceLong to a negative value).

Now there are several fundamental issues with the funding rate calculation which altogether will result in a flawed and manipulatable funding rate.

If it is somewhat possible to manipulate the funding rate to achieve the following state:

a) PnL increases balanceLong to result in tradingExpo becoming near zero
b) Funding fee is negative to result in tradingExpo becoming negative while balanceVault is smaller than tradingExpo

This can result in a state where tradingExpo becomes negative which then in turn results in balanceVault to become negative which then in turn results in tradingExpo to become negative:

```
// this can happen if the funding is larger than the remaining balance in the long side after applying PnL
// test case: test_assetToTransferZeroBalance()
if (tempLongBalance < 0) {
tempVaultBalance += tempLongBalance;
tempLongBalance = 0;
```

	} *// this can happen if there is not enough balance in the vault to pay the bad debt in the long side, for* *// example if the protocol fees reduce the vault balance* *// test case: test_funding_NegLong_ZeroVault()* *if (tempVaultBalance < 0) {* *tempLongBalance += tempVaultBalance;* *tempVaultBalance = 0;* *}* *// TODO: remove safe cast once we're sure we can never have negative balances* *longBalance_ = tempLongBalance.toUint256();* *vaultBalance_ = tempVaultBalance.toUint256();* Due to the issue: “EMA incorrectly uses funding from wrong time period”, this is a scenario which can happen in practice.
Recommendations	We recommend the following steps: a) Fixing the EMA/funding calculation. b) Applying extensive fuzz tests to ensure this edge-case can never happen in practice. Additionally, fuzzing should also be applied in the scenario of bad-debt liquidations.
Comments / Resolution	

Issue_75	Bonus application will alter imbalance towards the long side and can DoS the transaction
Severity	**High**
Description	Whenever a liquidation is happening and consecutively the Rebalancer is triggered (which is the case if the vault dominance becomes 6% or more), a bonus is deducted from the vault side and added to the Rebalancer. This bonus is calculated as follows: > remainingCollateral * _rebalancerBonusBps / divisor It is now important to understand that the whole call would simply return early in the case where the vault imbalance does not trigger the Rebalancer. Furthermore, it is important to understand that this bonus application decreases the vault dominance after the previously mentioned check. In certain scenarios, this bonus can become so large that the vault dominance already drops below 5.5% and therefore no rebalance action is needed. This case is specifically handled within the _calcRebalancerPositionTick function as follows: *// check that the target is not already exceeded* *if (cache.tradingExpo >= targetTradingExpo) {* *return Constants.NO_POSITION_TICK;* *}* A problem however occurs because this will result in a revert within the _triggerRebalancer function: *// make sure that the rebalancer was not triggered without a sufficient imbalance* *// as we check the imbalance above, this should not happen*

	if (tickWithoutLiqPenalty == Constants.NO_POSITION_TICK) { *revert IUsdnProtocolErrors.UsdnProtocolInvalidRebalancerTick();* *}* Thus the whole process will result in a DoS.
Recommendations	Consider refactoring the function in an effort to properly reflect the bonus deduction into the tick calculation, while ensuring that a position is in fact opened and the bonus is not lost. The simplest solution would be to create a position with the minimum leverage and accept the fact that the vault dominance will be slightly below 5.5%.
Comments / Resolution	

Issue_76	Special contract state can result in lower profit for newly created positions
Severity	**Medium**
Description	As we have highlighted multiple times in the report, it is theoretically possible to achieve a contract state where _balanceLong is larger than _tradingExpo due to funding application. If a new position is created, this position will have its own tradingExpo which is used to determine the PnL. In such a situation, the contract state would negatively affect the overall trading expo (because it contributes a net-negative). Therefore, if the price increases, the PnL calculation will be less impactful because the formula is as follows: balanceLong + (tradingExpo*priceDiff/newPrice) which means that tradingExpo is lower due to the previously explained contract state. Even more highlighting is the following code-section: *// if balanceLong == totalExpo or the long trading expo is negative (theoretically impossible), the PnL is* *// zero* *// we can't calculate a proper PnL value if the long trading expo is negative because it would invert the* *// sign of the amount* *if (balanceLong >= totalExpo) {* *return balanceLong.toInt256();* *}* this means if tradingExpo is negative and a position with a low amount/leverage is created, this position will eventually not experience

	any profit.
Recommendations	Consider keeping a reasonable minimum leverage, ensuring that _balanceLong is always smaller than _totalExpo.
Comments / Resolution	

Issue_77	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. While this, in itself, is correct for any PnL application **(see: "Appendix: Pnl calculation & liquidation price")**, the constant accumulation via funding should very well increase a position's long size above the total

	position exposure. *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_78	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. Furthermore we need to understand that the user can basically choose any oracle to fetch the price from, depending on which initial situation is being triggered (init/validation/validation after 20 minutes). 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	

Issue_79	Rebalancer reward is incorrectly paid out if rebalancer has not been triggered
Severity	**Medium**
Description	Whenever the _triggerRebalancer function is invoked, this will automatically mark the following variable as true: *data.rebalancerTriggered = true;* There are however some scenarios where the rebalancer is not triggered. Example: *if (address(rebalancer) == address(0)) {* *return (longBalance_, vaultBalance_);* *}* In that scenario, rewards for this interaction are still paid out. This issue was already found by the Smardex team during the audit.
Recommendations	Consider paying out rewards only if the rebalancer was triggered.
Comments / Resolution	

Issue_80	Liquidations can be triggered while preventing rebalancer bonus
Severity	**Medium**
Description	The Rebalancer is only triggered once all liquidatable positions have been liquidated. If not all positions have been liquidated, the rebalancer is not triggered. This fact can be abused by users to intentionally liquidate positions while keeping one leftover liquidatable position. This will result in the Rebalancer not receiving the bonus for these liquidated positions. **Illustrated:** a) There are 5 outstanding liquidatable ticks b) Normally, all these ticks would be liquidated and the rebalancer would get the bonus for all ticks The issue: A user can intentionally only liquidate 4 ticks while keeping the 5th tick pending, the bonus of these four ticks would not flow towards the Rebalancer in that scenario.
Recommendations	Consider triggering the Rebalancer after each liquidation. This will not automatically result in the rebalancer receiving a bonus each time because of the imbalance threshold. Eventual side-effects must be considered if the Rebalancer is triggered after each liquidation such as eventual interferences with the Rebalancer positions itself.
Comments / Resolution	

Issue_81	Users can forcefully imbalance the protocol by liquidating not all ticks and profiting from a high negative funding rate
Severity	**Low**
Description	After ticks have been liquidated, the protocol will most likely experience a (large) imbalance, which means that the funding rate can be very aggressively negative for some time (until the rebalancer is triggered). However, the Rebalancer is only triggered if all liquidatable positions have been liquidated. If there are outstanding liquidations, the rebalancer will not be triggered: *if (!isLiquidationPending_ && liquidationEffects.liquidatedTicks > 0) {* *...* *}* On the first sight, this doesn't seem like a big problem because before any new actions can be initiated, the protocol state will be updated and eventually all liquidatable positions will be triggered. It will however expose a problem if there are no actions/liquidations for a certain time because that means the protocol will stay imbalanced for some time and users can profit from a negative funding rate.
Recommendations	Consider triggering the Rebalancer under all circumstances after ticks have been liquidated. This may require some refactoring within the rebalancer trigger, similar to checks like this already existing one: *// if the position value is less than 0, it should have been liquidated but wasn't* *// interrupt the whole rebalancer process because there are pending liquidations*

	if (realPositionValue < 0) { *return (cache.longBalance, vaultBalance_);* *}* For the scenario where the rebalancer position itself is already liquidatable.
Comments / Resolution	

Issue_82	Rebalancer interaction does not ensure that protocol becomes balanced
Severity	**Low**
Description	The Rebalancer is always triggered whenever the vault *dominance becomes 6% or more*, while the target of the rebalancer is to open a position which decreases the vault dominance towards 5.5%. In most scenarios, there is always an open Rebalancer position which is included in the *previously calculated imbalance*. There are now two mechanisms which will result in the long side increasing its dominance: a) Any additional funds which are sitting unused in the rebalancer b) An increase of the leverage for the next position opening In such a scenario where there are no funds in the Rebalancer and the leverage is already 10x, there is no chance to offset the imbalance. This issue is highlighted with the scenario where the position amount becomes smaller than (_minLongPosition/10000).

Recommendations	Consider allowing for a higher leverage in that scenario.
Comments / Resolution	

Issue_83	Unupdated oracle timestamp may be used for contract state update
Severity	**Low**
Description	During opening and closing positions, the timestamp of the oracle update is used. This will expose a problem if the oracle is unupdated. currentPrice.timestamp is the timestamp of the oracle price update returned from the _getOraclePrice() function. This time can be lower than the current block.timestamp. currentPrice.timestamp is passed into _applyPnlAndFunding. This funding is later passed down and used to calculate the duration over which funding is charged: *elapsedSeconds = timestamp - s._lastUpdateTimestamp;* where timestamp is the timestamp of the latest price oracle update, not the current block.timestamp. Therefore funding is charged over the wrong duration. The duration should be the difference between the current timestamp and the timestamp of the latest state update. This way, the time over which funding is charged is the same as the duration where the user had an open position. Instead, funding is charged between the oracle time updates between a position being opened and closed.

	On the other hand, it is not possible to use the current timestamp because that would mean the contract state is updated up to the current timestamp using an unupdated price. This can have several side-effects including the prevention of liquidations etc.
Recommendations	Consider forcing users to update the contract state with a fresh oracle.
Comments / Resolution	

Issue_84	Target tradingExpo calculation favors vault side instead of long side
Severity	**Low**
Description	This issue is related to “Deposits (and Position closures) will never work after a rebalance” but not completely the same. Since liquidations result in a decrease of tradingExpo, it makes more sense to favor the long side during the target tradingExpo calculation, as this relieves the Rebalancer dominance. In the current implementation, the goal is to meet a 5.5% dominance towards the vault side. If this dominance increases to 6%, it will then quickly again be necessary to trigger the Rebalancer. If however the Rebalancer position is opened with such an expo that the long side will be in dominance, there is more wiggle room until the next Rebalancer execution is triggered.
Recommendations	Consider refactoring this calculation to shift the dominance towards the long side.
Comments / Resolution	

Issue_85	Dust is gifted to vault but not transferred in
Severity	**Low**
Description	Within the _triggerRebalancer function, there is a scenario which gifts the new positionAmount to the vault in the case where it is below (_minLongPosition/10000): *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(PositionId(Constants.NO_POSITION_TICK, 0, 0), 0);* *vaultBalance_ += data.positionAmount;* *return (cache.longBalance, vaultBalance_);* *}* As highlighted, the vault side is then increased by positionAmount, whereas positionAmount includes the already closed position value plus the amount of pending assets which are still in the Rebalancer. Due to the early return, the value in the Rebalancer is actually never transferred in, thus granting the vault side more funds than actually deposited. This issue has been rated as low because in the original setting this cannot happen due to the fact how _minLongPosition and _minAssetDeposit are determined. However, since it is technically incorrect, we still recommend fixing it.
Recommendations	Consider only increasing the vault side with the old position value: data.positionValue.

Comments / Resolution	

Issue_86	Malicious user can manipulate rebalance to bring imbalance towards long side if minimum leverage is changed at some point in the future
Severity	**Low**
Description	The only goal of the Rebalancer is to counter the impact of liquidations where the tradingExpo drops such that the protocol leans towards an imbalance in favor of the vault side. The mechanism to counter this is to simply open a long position, which then in reverse increases the tradingExpo to counter the imbalance. Whenever the long position is opened, it is bounded by three values: a) positionAmount b) minLeverage c) maxLeverage The _calcRebalancerPositionTick function is then calculating the ideal tickWithoutPenalty to achieve the desired tradingExpo. In such a scenario, where positionAmount is incredibly large, the imbalance could shift towards the tradingExpo, potentially causing a larger imbalance than previously existing. This works as long as the minimum leverage is kept as 10000000010000000000000, since it is impossible to have such a large amount of deposits If however, the minimum leverage is increased, it may happen that the

	a smaller amount of wstETH deposits will already push the imbalance towards the long side, exceeding the long imbalance threshold.
Recommendations	For now, we do not recommend any change. However, if it is planned to increase the minimum leverage, this should be kept in mind.
Comments / Resolution	

Issue_87	Liquidation reward payout will alter the imbalance after protocol has been rebalanced
Severity	**Low**
Description	The Rebalancer has the task to rebalance the protocol such that the vault side becomes 5.5% in favor towards the long side. After a rebalance has happened, liquidation rewards are transferred out which then again decrease the vault dominance by a small value, which then alters the imbalance and pushes it away from the target imbalance.
Recommendations	In our opinion, no change is needed because the decrease is negligible. However, if desired to fix: Consider paying out the liquidation reward before the protocol has been rebalanced. This can be trivially done by checking if the protocol should be rebalanced before the payout (including potential payout impact), then paying out the reward and only calculating the targetLongExpo after the payout.

Comments / Resolution	

Issue_88	Usage of getClosestTickAtPrice may favor users instead of protocol
Severity	**Low**
Description	Whenever ticks are liquidated, the first step in the liquidation process is to fetch the corresponding tick for the current price. This is done using the getClosestTickAtPrice function. This function fetches - as the name indicates - the closest tick for a current price, instead of the next lower tick for the current price. Example: 2799999999999999999 getClosestTickAtPrice = 79378 getTickAtPrice = 79377 In the latter scenario, the protocol is favored because the liquidation tick is lower and thus liquidations are happening earlier. (Leverage is decreased). At Bailsec, we are of the opinion that every operation in a smart contract should be against the favor of the user and towards the protocol, simply due to the reason to give users as little flexibility as possible. Furthermore, in our opinion it should accommodate for tickSpacing to round down the liquidation price. This would then result in a possible disadvantage for users but at the same time it can prevent unforeseen issues when converting a tick which is not a multiple of 100 to the corresponding index, later in the _updateStateAfterLiquidation and _findHighestPopulatedTick functions.

Recommendations	Consider using the getTickAtPrice function instead getClosestTickAtPrice.
Comments / Resolution	

Issue_89	Maximum leverage is impossible to reach due to usage of adjustedPrice
Severity	**Informational**
Description	The leverage is calculated using the adjustedPrice which includes downscaling of the price and a nominal fee: > adjustedPrice / (adjustedPrice - liqPriceNoPenalty) This makes it hard for users to actually predict the price parameter that should be provided with the function call, because the adjustedPrice will always be a bit higher than the real price and if the transaction stucks in the mempool, the price can change as well. If users for example provide a desiredLiqPrice of 2754 (includes penalty), that will translate to a liqPriceNoPenalty of 2700, which aims for 10x leverage if the price is 3000. However, if now adjustedPrice is 3010 instead of 3000, the following leverage will be calculated: > 10e21 * 3010e18 / (3010e18 - 2700e18) > 9.709e22 As one can see, the target of 10x is not reached. This issue is also present within other parts of the codebase, notably in

	the case where the position is validated and the maxLeverage was exceeded, it will calculate the liquidation price based on maxLeverage and the price including the fee.
Recommendations	We currently do not recommend any change at this point, however, this should be kept in mind.
Comments / Resolution	

Issue_90	Low _tickSpacing will result in large gas consumption during _liquidatePositions
Severity	**Informational**
Description	While the Smardex team openly communicated that the protocol will be deployed with _tickSpacing = 100, it is not hardcoded within the constructor. In the scenario where a lower _tickSpacing will be chosen, it will result in expensive liquidation calls.
Recommendations	Consider deploying the contract with _tickSpacing = 100.
Comments / Resolution	

Issue_91	Rebalancer position can be slightly higher than 10x leverage
Severity	**Informational**
Description	In the scenario where tradingExpoToFill is minimally lower than highestUsableTradingExpo, that means that the position can have a leverage up to 9.99999x, which would then also reflect a value near the desired imbalance target. Furthermore, during the tick calculation via the getEffectiveTickForPrice, the determined tick is rounded down which can result in a price up to 1% less. If we now consider the last if-clause, the tick is increased by 1%, which can then result in a slightly higher leverage (if the previous price decrease due to rounding was less than 1%).
Recommendations	No change is needed, we recommend keeping this edge-cases in mind.
Comments / Resolution	

ProtocolSetters

The ProtocolSetters library is invoked by the UsdnProtocol contract and incorporates the logic for parameter changes. Governance can change the following parameters:

- _oracleMiddleware
- _liquidationRewardsManager
- _rebalancer
- _minLeverage
- _maxLeverage
- _validationDeadline
- _liquidationPenalty
- _safetyMarginBps
- _liquidationIteration
- _EMAPeriod
- _fundingSF
- _protocolFeeBps
- _positionFeeBps
- _vaultFeeBps
- _rebalancerBonusBps
- _sdexBurnOnDepositRatio
- _securityDepositValue
- _feeThreshold
- _feeCollector
- all imbalance limits
- _targetUsdnPrice
- _usdnRebaseThreshold
- _usdnRebaseInterval
- _minLongPosition

Privileged Functions

- none

Issue_92	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	

Issue_93	Change of _liquidationPenalty will inadvertently result in users accidentally getting liquidated early
Severity	**High**
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. Furthermore, there is absolutely no way to prevent this scenario, users would inadvertently suffer from this.
Recommendations	Consider executing a check that there are no initiated long positions before the _liquidationPenalty is changed.
Comments / Resolution	

Issue_94	Change of _rebalancerBonusBps will be applied on hindsight
Severity	**Low**
Description	The _rebalancerBonusBps determines how much of the remainingCollateral from the liquidated tick will be assigned towards the Rebalancer. A change of this variable will affect liquidations which should have happened but have not been triggered thus far, resulting in a change in hindsight.
Recommendations	Consider ensuring that there are no outstanding liquidations before this variable is changed.
Comments / Resolution	

Issue_95	Change of _protocolFeeBps will be applied in hindsight
Severity	**Low**
Description	The _protocolFeeBps variable determines how much of the funding fee is allocated towards the protocol. In a state where the protocol has not been updated for a long time, the change of _protocolFeeBps will have an impact in hindsight, changing the fee application.
Recommendations	Consider updating the protocol before the change of _protocolFeeBps.
Comments / Resolution	

Issue_96	Change of _vaultFeeBps and _positionFeeBps can result in discrepancy between initiation and validation phase
Severity	**Low**
Description	Whenever one of both parameters is changed after a user has initiated an action, this will inherently result in a change of the validation outcome, with no possibility for users to revert their previous initiated action.
Recommendations	Consider announcing such a change beforehand.
Comments / Resolution	

Issue_97	Change of _validationDeadline can suddenly disrupt position initiations
Severity	**Low**
Description	The _validationDeadline is initially set to 90 minutes and then used throughout the architecture on the following occasions: ProtocolCore._removeBlockedPendingAction ProtocolCore._getActionablePendingActions If the _validationDeadline is suddenly reduced, this means unaware users might lose their security deposit or in the worst case their full deposit (this would need governance interaction).
Recommendations	A code-focused fix would be to incorporate the actual _validationDeadline within the init action and then only fetch and use this value. Due to the modularity of the architecture and the large function flow of these calls, this must be very carefully implemented. Optionally, another idea might be to communicate such a change with the community with a sufficient grace period (ie. 3 days before the change is being made).
Comments / Resolution	

Issue_98	Change of _validationDeadline can result in unexpected loss of securityDeposit
Severity	**Low**
Description	If the _validationDeadline is decreased without previous announcement, it can happen that validators lose their anticipated securityDeposit because other users can now validate an actionable action.
Recommendations	Consider announcing such a change beforehand.
Comments / Resolution	

ProtocolVault

The ProtocolVault library is a fundamental part of the modular USDN architecture which exposes several important functions. First of all it exposes important view-only functions such as the current USDN price, deposit and withdrawal estimations and other functions. Secondly, the ProtocolVault library handles the full internal logic for the protocol initiation which means:

a) Initial Deposit Creation
b) Initial Position Opening
c) Initial Imbalance Check

In the scenario where an initiated action has not been validated within 2.5 hours, this action can be removed by governance either with or without cleaning up the contract state.

Furthermore, this library exposes multiple helper functions for vault interactions and the USDN rebase logic.

Appendix: USDN Rebase

The USDN token is a rebase token with steadily increasing supply. The goal of this token is to stay in a range between 1.0084 and 1.009 USD. The token value is determined using the vault balance, the current wstETH price and the USDN supply via the following formula:

balanceVault * wstETHPrice / totalUSDNSupply

Therefore, there are four main scenarios which will result in a price increase/rebalance:

a) balanceVault increases via liquidations
b) balanceVault increases via positive funding rate application
c) balanceVault increases via PnL applications
d) wstETH price increases

Whenever the token price crosses the threshold of 1.009 USD, a rebalance is executed with the goal to increase the totalUSDNSupply to such a value which results in the target price of 1.0084 USD. This is done as follows:

a) Calculate the targetSupply by applying the targetPrice on the current vaultBalance:

vaultBalance * wstETHPrice / targetPrice

b) Calculate the new divisor to reflect the desired target supply

usdnTotalSupply * oldDivisor / usdnTargetSupply

Such a rebase is always triggered during the general protocol state update (PnL/funding/liquidations).

Privileged Functions

- none

Issue_99	Usage of removeBlockedPendingActionNoCleanup will manipulate the protocol state
Severity	**Governance**
Description	The ProtocolVault library exposes two functions that allow for removing initiated actions which have not been validated within 2.5 hours after the initiation. Such a state is likely only reached if there is a serious issue with the protocol because under normal circumstances, initiated actions are automatically validated during the normal business logic plus there are additional incentives to validate these manually via the validateActionablePendingAction function which transfers the securityDeposit amount to the caller. These two functions are namely: a) removeBlockedPendingAction b) removeBlockedPendingActionNoCleanup

	The first function simply reverses the initiated process versus the second function does not take care of the storage amendment. For the second function, this can become a serious issue after a certain time because these positions are still considered as “initiated” by the protocol, which then in turn influences the overall protocol state, such as permanently keeping an increased _pendingBalanceVault variable, as one of many examples. Obviously this will also result in funds being permanently stuck.
Recommendations	Since this is a design-choice and will only be invoked upon emergency cases, we do not recommend applying a change here.
Comments / Resolution	

Issue_100	Off-by-one error during _checkInitImbalance
Severity	**Low**
Description	The _checkInitImbalance function ensures that the initial deposit and position opening will not result in imbalances towards the vault or long side. This is done using the following checks: *if (imbalanceBps > depositLimit) {* *revert IUsdnProtocolErrors.UsdnProtocolImbalanceLimitReached(imbalanceBps);* *}* *if (imbalanceBps > openLimit) {* *revert IUsdnProtocolErrors.UsdnProtocolImbalanceLimitReached(imbalanceBps);* *}* Compared to all other imbalance checks in the protocol, this only reverts if the imbalance exceeds the limit, versus in all other checks it already reverts if the imbalance meets the limit: *if (imbalanceBps >= depositExpoImbalanceLimitBps) {* *revert IUsdnProtocolErrors.UsdnProtocolImbalanceLimitReached(imbalanceBps);* *}*
Recommendations	Consider staying consistent in the imbalance check.
Comments / Resolution	

Issue_101	Inconsistency between protocol initiation and long position opening
Severity	**Low**
Description	Whenever a standard long position is opened, the provided input parameter desiredLiqPrice already includes the penalty. The position is then stored at this tick and the totalExpo is calculated using the tick without the penalty. Contrary to that, upon protocol initialization, the input parameter desiredLiqPrice does not include the penalty. The penalty is added on top of the provided input parameter and the position is stored on this tick then. This is a small inconsistency in the protocol design which does not have any negative impact. However, generally inconsistencies should be avoided at all costs.
Recommendations	Consider being strict and following one logic consistently.
Comments / Resolution	

Issue_102	Violation of checks-effects-interactions pattern
Severity	**Informational**
Description	[Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes). https://github.com/SmarDex-Ecosystem/usdn-contracts/blob/ff2b96fefc6ab41bb2adccef430c1f2f19e29dda/src/UsdnProtocol/libraries/UsdnProtocolVaultLibrary.sol#L173 https://github.com/SmarDex-Ecosystem/usdn-contracts/blob/ff2b96fefc6ab41bb2adccef430c1f2f19e29dda/src/UsdnProtocol/libraries/UsdnProtocolVaultLibrary.sol#L206
Recommendations	Consider executing external calls after any storage changes.
Comments / Resolution	

ProtocolConstants

The ProtocolConstants contract exposes all important variables such as decimal denomination for different calculations and other fundamental values.

Privileged Functions

- none

No issues found.

Core Contracts

The following Core contracts form the entry point for the USDN protocol. These contracts expose all necessary interfaces to interact with the protocol while outsourcing most or all of the logic to the above library contracts.

UsdnProtocol

The UsdnProtocol contract exposes all necessary functions for the governance body to amend the protocol state, such as changing the OracleMiddleware contract, setting the liquidation penalty or adjusting rebalance limits.

It is the only non-abstract contract and inherits all other contracts, which together, form the deployable base.

Privileged Functions

- transferOwnership
- renounceOwnership
- acceptOwnership
- setOracleMiddleware
- setLiquidationsRewardsManager
- setRebalancer

- setMinLeverage
- setMaxLeverage
- setValidationDeadline
- setLiquidationPenalty
- setSafetyMarginBps
- setLiquidationIteration
- setEMAPeriod
- setFundingSF
- setProtocolFeeBps
- setPositionFeeBps
- setVaultFeeBps
- setRebalancerBonusBps
- setSdexBurnOnDepositRatio
- setSecurityDepositValue
- setFeeThreshold
- setFeeCollector
- setExpoImbalanceLimits
- setTargetUsdnPrice
- setUsdnRebaseThreshold
- setUsdnRebaseInterval
- setMinLongPosition

No issues found.

UsdnProtocolActions

The UsdnProtocolActions contract forms the entry contract for all important actions, namely:

- Deposit initiation
- Deposit validation
- Withdraw initiation
- Withdraw validation
- Opening initiation
- Opening validation
- Closing initiation
- Closing validation
- Liquidation
- Actionable validation
- Position transfer

The whole logic was outsourced to the library module.

Privileged Functions

- none

No issues found.

UsdnProtocolCore

The UsdnProtocolCore contract exposes the entry function for contract initialization as well as view-only functions for different core purposes.

Moreover it exposes governance functions for the removal of pending actions which are more than 2.5 hours in the system.

The whole logic was outsourced to the library module.

Privileged Functions

- removeBlockedPendingAction
- removeBlockedPendingActionNoCleanup

No issues found.

UsdnProtocolLong

The UsdnProtocolLong contract exposes various view-only functions which are corresponding to long positions.

The whole logic was outsourced to the library module.

Privileged Functions

- none

No issues found.

UsdnProtocolStorage

The UsdnProtocolStorage contract represents (as the name already indicates) the storage layout for the core protocol. It exposes all necessary storage variables and is inherited as part of the core module. Furthermore it exposes the constructor and most initial settings for the contract deployment.

Privileged Functions

- none

No issues found.

UsdnProtocolVault

The UsdnProtocolVault contract exposes view only functions which are corresponding to the vault module. Furthermore it exposes governance functions to remove pending actions from a specific validator.

The whole logic was outsourced to the library module.

Privileged Functions

- removeBlockedPendingAction
- removeBlockedPendingActionNoCleanup

No issues found.

Utils

InitializableReentrancyGuard

The InitializableReentrancyGuard is a utility contract that facilitates a reentrancy guard paired with a safety mechanism which ensures that the protocol is initialized. The initializedAndNonReentrant modifier is applied upon all important functions within the UsdnProtocolActions contract.

Initially, the status is marked as UNINITIALIZED, once the contract is then initialized, the status is marked as NOT_ENTERED and whenever a function is being executed the status is temporarily set to ENTERED and after successful execution then set back to NOT_ENTERED.

Privileged Functions

- none

No issues found.